CRITICAL PROBLEMS IN VERY LARGE SCALE COMPUTER SYSTEMS

Semiannual Technical Report for the Period

April 1, 1990 to September 30, 1990

Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Anant Agarwal	(617) 253-1448
William J. Dally	(617) 253-6043
Srinivas Devadas	(617) 253-0454
Thomas F. Knight, Jr.	(617) 253-7807
F. Thomson Leighton	(617) 253-3662
Charles E. Leiserson	(617) 253-5833
Jacob K. White	(617) 253-2543

[1]This research was sponsored by Defense Advanced Research Projects Agency (DoD), through the Office of Naval Research under Contract No. N00014-87-K-0825.

Contents

1 Research Overview

The research vehicle for this contract is the largest possible computer that can be conceived for the mid to late 1990's. We call this machine an "American Resource Computer" or "ARC." We imagine this machine to occupy several floors of a building. The nation could probably only afford one or two ARC's. The machine will be used to solve large-scale scientific problems having both military and civilian applications.

This investigation addresses the hardware technology, software techniques, algorithms, communications, processing elements, and applications. The study is determining the plausibility (not feasibility) of the machine. The technical challenges of such a machine serve as our guiding stimulus for the research carried out and reported here.

The chip technology that will be available for an ARC is consistent with the following parameters, assuming a CMOS process with $\lambda = 0.125$ microns.

Size:	10m × 10m × 10m
FLOPS:	10^{15}
Bits:	10^{15}
Cost:	\$1–2 billion
Processors:	4 billion
Number of chips:	10 million
Clock:	200 MHz
Power:	100 MW (10W/chip)
Bisection bandwidth:	10^{16} bits/sec
Total node bandwidth:	10^{19} bits/sec
Component reliability:	1 hour MTBF
System reliability:	???

Research is required to overcome the numerous hurdles to making an ARC feasible. Of the issues to be faced, the most problematic is system reliability. A mean time to failure of 10^5 hours is plausible, but significant research must be done to achieve this goal economically.

Progress in the various research areas are highlighted in the forthcoming sections.

2 Circuits

Kevin Lam, Larry Dennision, and William Dally have developed a CMOS transceiver circuit that permits high-speed digital signals to be transmitted in both directions over a single wire simultaneously [112]. The circuit is similar in concept to the *hybrid* used in telephone systems to convert a four-wire circuit to a two-wire circuit. A current source driver sums the transmitted (forward) signal onto a transmission line. A clocked differential receiver (a sense-amp) subtracts the forward signal from the superposition of forward and reverse signals to recover the received (reverse) signal. Careful delay matching is required to match the phase of the transmitted signal and the subtracted signal at high frequencies.

Operating in a single direction, the circuit demonstrates techniques that have recently been developed by Dally's group for low-voltage CMOS signalling. The transmitted signal swing is 0.5V. The use of a current source driver and a current-sensing receiver isolate the signal from power supply noise permitting reliable signaling in noisy environments. The clocked receiver dissipates no static power, a significant advantage over power-hungry cascode receivers.

SPICE simulations of the circuit using 2μ CMOS parameters indicate that it operates reliably at 100Mbits/s. Prototype tranceivers have been fabricated in 2μ CMOS through MOSIS and are currently under evaluation.

Alexander Ishii has also been working with Thomas Knight on an implementation of a self-terminating, digitally-controlled, and ECL-compatible output pad driver for high-speed integrated circuits. By automatically series-terminating driven lines with their characteristic impedances, the driver realizes speed, power, and noise improvements over conventional designs. The design has not yet been fabricated, but simulations indicate that data-transition rates in excess of 100MHz are possible.

3 Processing Elements

Anant Agarwal and his students have been working on processor design in connection with the Alewife multiprocessor system. When the system cannot avoid a remote memory request and is forced to incur the latency of the communication network, the Alewife processors try to tolerate the latency by rapidly scheduling another process. Alewife can also tolerate synchronization latencies through the same context switching mechanism. Because context switches are forced only on memory requests that require the use of the interconnection network and on synchronization faults, the processor achieves high single-thread performance. They have designed a new processor architecture called APRIL that can rapidly switch between processes. This fast context-switch is achieved by caching four sets of register frames on the processor to eliminate the overhead of loading and unloading the process registers. The SPARCLE processor is being implemented jointly with LSI Logic and SUN Microsystems through modifications to an existing SPARC design. SPARCLE will switch between processes in 11 cycles and clock at 33MHz. In the present period LSI has completed the RTL specification of the part, and the team at MIT is modifying the test system used by LSI to include their model of the memory system.

4 Communications Topology and Routing Algorithms

William Dally and Hiromichi Aoki have been developing adaptive routing strategies that employ virtual channels. The use of adaptive routing in a multicomputer interconnection network improves network performance by making use of all available paths and provides fault tolerance by allowing messages to route around failed channels and nodes. Dally and Aoki have developed two deadlock-free adaptive routing algorithms. Both algorithms allocate virtual channels using a count of the number of *dimension reversals* a packet has performed to eliminate cycles in resource dependency graphs. The *static algorithm* eliminates cycles in the network channel dependency graph. The *dynamic algorithm* improves virtual channel utilization by permitting dependency cycles and instead eliminating cycles in the packet wait-for graph. These algorithms are particularly well suited to VLSI implementation. They require less control storage than table-driven routing algorithms and less data storage than packet-based routing algorithms.

They have conducted a simulation study to evaluate these two adaptive routing algorithms. For non-uniform traffic patterns, these algorithms improve network throughput by a factor of three compared to deterministic routing using the same number of virtual channels. The dynamic algorithm gives better performance at moderate traffic rates but requires source throttling to remain stable at very high (over 100% network capacity) traffic rates. Both algorithms allow the network to gracefully degrade in the presence of faulty channels. With the dynamic algorithm, a failure of 8 percent of the network channels (38 channels) in a 16-ary 2-cube increases latency on average by a factor of 2.2.

They are currently investigating implementation strategies for the dynamic adaptive routing algorithm.

William Dally and Larry Dennison have started a project to develop an architecture for a programmable, general-purpose network router. Such a router could be programmed to support many different topologies, routing agorithms, and flow-control strategies. In contrast, conventional routers are hard-wired to support a single network topology, routing algorithm, and flow-control strategy.

During the past six months, Dally and Dennison have sketched a high-level architecture for such a router. Input controllers in this proposed router execute a routing program to select the next channel of a route. A hard-wired virtual-channel flow control mechanisms can be used by a routing program to implement blocking, buffering, dropping, or misrouting flow control. They are currently investigating issues involving instruction set design, collection and encoding of channel status information, and details of the underlying flow control mechanism.

Dally and Dennison have also begun a study to develop methods for constructing highly-reliable, large-scale interconnection networks. Their first step has been to study end-to-end and link-level retry protocols that ensure reliable exactly-once message delivery. Under certain assumptions about message locality they have been able to show that the number of buffers required for a reliable end-to-end protocol can remain constant as the network scales. They are currently evaluating these protocols using statistical models of network traffic. They are also developing fault models for interconnection networks and investigating methods for fault detection and containment.

William Dally's group has also been investigating methods for improving the physical efficiency of multicomputer interconnection networks. An efficient network is one that makes the best possible use of all available resources: chip area and wiring volume.

Over the past year they have developed and evaluated the express-cube topology [22]. Express cube networks simultaneously approach the physical and information-theoretic limits of network performance. For messages going long distances latency can be made arbitrarily close to the physical speed of light limit by adding appropriate express channels. For short distances, latency grows logarithmically achieving the information theoretic bound. By adding multiple express channels, throughput can be increased to use all available wiring volume. Recently, they have been investigating practical methods for implementing express cubes.

To achieve physical bounds on network throughput, flow control strategies are required that permit the duty factor of network channels to approach 100 percent. Conventional queueing systems saturate at between 25 and 50 percent capacity even on uniform loads because of resource coupling between buffers and channels. A flow-control method has recently been developed based on virtual channels [25] that decouples buffer allocation from channel allocation enabling channel duty factors to approach 100 percent.

Network throughput can be increased by dividing the buffer storage associated with each network channel into several virtual channels. Each physical channel is associated with several small queues, virtual channels rather than a single deep queue. The virtual channels associated with one physical channel are allocated independently but compete with each other for physical bandwidth. Virtual channels decouple buffer resources from transmission resources. This decoupling allows active messages to pass blocked messages using network bandwidth that would otherwise be left idle. Simulation studies show that, given a fixed amount of buffer storage per link, virtual-channel flow control increases throughput by a factor of 3.5, approaching the capacity of the network.

Their recent work on virtual-channel flow control has concentrated on a study of scheduling algorithms for allocating channel bandwidth among competing buffers and on extending the technique to work with the adaptive routing algorithms described above.

5 Systems Software

Anant Agarwal and his students have continued their work on automatic locality management in large-scale multiprocessors. A prototype multiprocessor system called Alewife is being designed to incorporate these methods. The past six months have seen substantial progress in several areas including runtime systems, compiler technology, and scalable cache coherence methodology; these developments are described below, after overviewing the bigger problem they are trying to solve.

Alewife is a large-scale multiprocessor with distributed shared memory. Reflecting the physical constraints of three-dimensional space, the machine uses a cost-effective mesh network for communication. This type of architecture scales in terms of hardware cost and allows the exploitation of locality. Unfortunately, the non-uniform communication latencies make such machines hard to program because the onus of managing locality invariably falls on the programmer. The goal of the Alewife project is to discover and to evaluate techniques for automatic locality management in scalable multiprocessors in order to insulate the programmer from the underlying machine details. Their approach to achieving this goal employs techniques for *latency minimization* and *latency tolerance*.

Agarwal's group has developed, implemented, and evaluated several mechanisms in the Alewife compiler, runtime system, and hardware that will cooperate in enhancing communication locality, thereby reducing communication latency and required network bandwidth.

- Shared-data caching in Alewife is an example of a hardware method for reducing communication traffic. Agarwal's group has recently found a new solution to the cache coherence problem in scalable multiprocessors, called LimitLESS directories. This scheme has been implemented in the Alewife simulator, ASIM. Simulations on ASIM show that the LimitLESS cache coherence protocol realizes the performance of the full-map directory protocol, with the memory overhead of a limited directory, but without excessive sensitivity to software optimization. The LimitLESS scheme implements a small set of pointers in the memory modules, but when necessary, the scheme allows a memory module to interrupt the processor for software emulation of a full-map directory. Since this new

coherence scheme is partially implemented in software, it can work closely with a multiprocessor's compiler and run-time system.

- They have developed a near-neighbor scheduling method coupled with dynamic task partitioning for minimization of communication latency. The design of the scheduling algorithms had to solve several interesting problems relating to rapid context-switching in the processors, and live-lock in the synchronization handlers. The scheduler has been implemented and runs on ASIM. On several test programs, including Multigrid, ASIM simulations indicate that locality-based scheduling can improve the performance of even modest-sized (16-64 processors) network-based multiprocessors.

- They have defined a new intermediate form for the compilation of parallel languages to distributed memory machines. The intermediate representation is called WAIF, which stands for Waif is Alewife's Intermediate Form. Because WAIF includes information on data dependences and volume of communication, it will aid in automatic partitioning and placement of data and processes, to minimize communications. A front end for a futures-based parallel dialect of C and for Mul-T are being written.

Other work by Agarwal and his students has involved developing large numeric and symbolic application test suites for testing their ideas. Two versions of the multigrid relaxation algorithm now available, one using static allocation of data and processes to processors, and the other using dynamic allocation.

His group has also come up with a new notion of scalability in parallel machines. Although scalability is an important consideration in the design of parallel computer architectures, a commonly-accepted, precise definition for scalability does not exist. They have a new definition based on the notion of "asymptotic speedup." This definition focuses on the mapping between a given algorithm's communication behavior and the communication paths provided by a given architecture. Due to large differences in communication requirements for different interesting algorithms, their definition is made with respect to both a given architecture and a given algorithm. The effects of physical constraints imposed by the three-dimensionality of space and fundamental limits on communication speeds are also considered.

They have recently shown that trace-driven simulations of caches and memory systems coupled with an analytical model of an interconnection network can yield accurate estimates of multiprocessor performance. Although trace-driven simulation has been used heavily in the design of multiprocessors, they believe this is the first validation of this important technique for multiprocessor studies.

6 Algorithms

Prof. Leighton continued work on the design of fast and fault-tolerant architectures and algorithms for parallel computation. Highlights of the work during the past six months are summarized below.

Continued progress has been made on the development of the multibutterfly. As he has reported previously, the multibutterfly appears to be an exceptionally promising network for fast highly-fault-tolerant message routing. Recent work has focussed on using multibutterfly-like architectures to route messages in a cut-through and nonblocking fashion. (Previous work focused more on store-and-forward applications.) The new experimental data obtained for multibutterfly networks demonstrates that the basic method is highly successful for all routing models tested, providing significant decreases in message delay and message blocking for randomly generated traffic, as well as a very high degree of fault-tolerance.

Leighton has also developed smaller, faster, and more fault-tolerant sorting networks. Working with Dr. Greg Plaxton (formally an MIT postdoc, and now a Professor at U.T. Austin), Professor Leighton has discovered an entirely new approach to the classical problem of building fast sorting circuits. Sorting circuits are used in a wide variety of applications, and have been the subject of much study during the past 3 decades. Most recently, they have been used for packet switching in telephone and data networks. Although $O(\log N)$-depth sorting networks were discovered in the early 1980's, they are highly impractical due to the (large) size of the constant hidden by the big O notation. As a consequence, Batcher's $\log N(\log N + 1)/2$-depth network (discovered in the mid-1960's) is still the circuit of choice for most applications. The new circuit discovered by Leighton and Plaxton uses randomization and sorts N numbers in $c \log N$ steps where c is probably less than 7.5, and experimentally is less than 4. Hence, the

circuit uses substantially less hardware and is faster than the Batcher circuit for problems of size 256 or more. The Leighton/Plaxton circuit has the disadvantage that the items to be sorted must be scrambled before they are sorted (the circuit works with very high probability for random inputs), but it has the advantage of being highly fault-tolerant. Experimental data indicates that a 1000-input 50-level circuit can sustain over 1000 random faults without losing reliability. The Batcher network, on the other hand, is not at all fault-tolerant.

In other work, Prof. Leighton also made substantial progress on the problem of computing with faulty arrays of processors, and on drawing graphs in the plane with high resolution.

Bruce Maggs, Derek Lisinski, and Tom Leighton have written a program to compare the performance of randomly-wired multistage networks (such as the multibutterfly) to more traditional non-randomly-wired multistage networks (such as the butterfly). The simulations show that in variety of routing contexts, including circuit-switching, packet-switching, and cut-through routing, randomly-wired multistage networks outperform non-randomly-wired networks constructed with equal hardware. In addition, the simulations show that multibutterfly networks can tolerate large numbers of faults with little degradation in performance.

During the past six months, James K. Park has been collaborating with Alok Aggarwal, Dina Kravets, and Sandeep Sen on a number of problems relating to Monge arrays. Aggarwal and Park have been studying the use of Monge arrays in solving economic lot-size problems arising in operations research. Aggarwal, Kravets, Park, and Sen have been investigating parallel alogorithms for searching in staircase-Monge arrays and the conversion of PRAM algorithms for searching in Monge arrays to algorithms for hypercubes and related interconnection networks.

Park has also been working on his doctoral thesis, titled "The Monge Array: An Abstraction and Its Applications." The thesis, a comprehensive study of Monge arrays and their applications, should be completed by January 1991.

Marios Papaefthymiou completed his master's thesis [80] under the supervision of Charles Leiserson. Papaefthymiou's thesis investigates problems on retiming and mixed-integer optimization. It presents efficient algorithms for optimal pipelining of combinational circuitry. It gives a characterization of the minimum feasible clock-period of a general circuit, in terms of the maximum delay-to-register ratio of the cycles in the circuit graph, that leads to more efficient algorithms for retiming. It describes the closed semiring structure of unit-delay circuitry retiming. The thesis also includes recent work on mixed-integer optimization. Specifically, it describes an $O(V^3 \lg V)$ algorithm for a mixed-integer optimization problem that arises in retiming.

Tom Cormen is working on two practical issues that arise in parallel computing. One issue is the interaction of context and parameter-passing in data-parallel computing. He is attempting to catalog how existing data-parallel languages treat this issue and recommend a robust solution to this linguistic problem. The second issue concerns virtual memory in data-parallel computing: how many disk operations and how many processor operations are required to perform common operations on parallel vectors when virtual processor ratios are so high that each physical processor can access only a small portion of its assigned data at one time?

Shlomo Kipnis continued his research on the organization of systems with bussed interconnections. During the past year he further explored the power of bussed interconnection schemes, interviewed for a research position at several laboratories, and completed his Ph.D. requirements. He finished his dissertation, entitled "Organization of Systems with Bussed Interconnections" [72], under the supervision of Professor Charles Leiserson. His thesis contains three research contributions.

In his thesis, Kipnis first explores the problem of efficiently permuting data stored in VLSI chips in accordance with a predetermined set of permutations. By connecting chips with shared bus interconnections, as opposed to point-to-point interconnections, he shows that the number of pins per chip can often be reduced. For example, he exhibits permutation architectures with $\sqrt{n}$ pins per chip that can realize any of the n cyclic shifts on n chips in one clock tick. When the set of permutations forms a group with p elements, any permutation in the group can be realized in one clock tick by an architecture with $O(\sqrt{p \lg p})$ pins per chip. When the permutation group is abelian, only $O(\sqrt{p})$ pins suffice. These results are all derived from a mathematical characterization of *uniform permutation architectures* based on the combinatorial notion of a *difference cover*. He also considers uniform permutation architectures that realize permutations in several clock ticks, instead of one, and shows that further savings in the number of

pins per chip can be obtained. This research represents joint work with Joe Kilian and Charles Leiserson (see [45]).

Next, Kipnis investigates priority arbitration schemes that use busses to arbitrate among n modules in a digital system. He focuses on distributed mechanisms that employ m busses, for $\lg n \leq m \leq n$, and use asynchronous combinational arbitration logic. A widely used distributed asynchronous mechanism is the *binary arbitration* scheme, which with $m = \lg n$ busses arbitrates in $t = \lg n$ units of bus-settling time. He presents a new asynchronous scheme — *binomial arbitration* — that by using $m = \lg n + 1$ busses reduces the arbitration time to $t = \frac{1}{2}\lg n$. Extending this result, he presents the *generalized binomial arbitration* scheme that achieves a bus-time tradeoff of the form $m = O(tn^{1/t})$ between the number of arbitration busses m, and the arbitration time t (in units of bus-settling time), for values of $1 \leq t \leq \lg n$ and $\lg n \leq m \leq n$. These schemes are based on a novel analysis of *data-dependent delays*. Most importantly, these schemes can be adopted with no changes to existing hardware and protocols; they merely involve selecting a good set of priority arbitration codewords. These results appeared in [46] and a patent application on the new arbitration schemes was filed by the MIT Technology Licensing Office.

Finally, Kipnis examines the performance of priority arbitration schemes under a *digital transmission line* bus model. This bus model accounts for the propagation time of signals along bus lines and assumes that the propagating signals are always valid digital signals. A widely held misconception is that in the digital transmission line model the arbitration time of the binary arbitration scheme is at most 4 units of bus-propagation delay. He formally disproves this conjecture by demonstrating that the arbitration time of the binary arbitration scheme is heavily dependent on the arrangement of the arbitrating modules in the system. He provides a general scenario of module arrangement on m busses, for which binary arbitration takes at least $m/2$ units of bus-propagation delay to stabilize. He also proves that for general arrangements of modules on m busses, binary arbitration settles in at most $m/2+2$ units of bus-propagation delay, while binomial arbitration settles in at most $m/4+2$ units of bus-propagation delay, thereby demonstrating the superiority of binomial arbitration for general arrangements of modules under the digital transmission line model. For linear arrangements of modules in increasing order of priorities and equal spacings between modules, he shows that 3 units of bus-propagation delay are necessary for binary arbitration to settle, and he sketches an argument that 3 units of bus-propagation delay are also asymptotically sufficient.

Alexander Ishii has begun work on a stand-alone timing-verification system for level-clocked VLSI circuits. The system implements and extends the timing-verification algorithms developed with Leiserson, and is intended to demonstrate how their formal results can be adapted to address a wide variety of high-performance circuit structures and engineering concerns.

Charles Leiserson led a group of researchers that included Bruce Maggs, Gregory Plaxton, Guy Blelloch of CMU, Steven Smith of Thinking Machines, and Marco Zagha of CMU to develop a fast sorting parallel sorting program. The work, sponsored in part by Thinking Machines, produced a fast implementation of the flashsort algorithm of Valiant and Reif. On a 64K-processor CM-2, their flashsort implementation sorts 5×10^6 64-bit keys in under 1 second, which, to their knowledge, makes it the fastest general-purpose sorting program ever reported.

7 Applications

Srinivas Devadas and his students have been continuing research in the areas of sequential logic synthesis, synthesis for testability of combinational and sequential circuits, test generation and formal verification.

Devadas' group has considered the synthesis of robustly path-delay-fault testable circuits and shown that a single property, *Equivalent Normal Form (ENF) reducibility*, allows us to unify previous results on robust delay-fault testability and multifault testability, as well as to prove new ones. They use the notion of ENF reducibility to show that a constrained version of a common area improving transformation namely, *algebraic resubstitution with complement* retains robust path-delay-fault testability. Thus, in addition to providing a comprehensive framework for understanding previous results, a more efficient means of synthesizing fully path-delay and gate-delay fault testable networks has been given. They have also used ENF reducibility to show that constrained algebraic resubstitution with complement retains multifault irredundancy [15].

They have also shown how a sophisticated orchestration of combinational synthesis for testability

approaches can result in logic-level implementations of large integrated circuit designs that are *completely robustly path-delay-fault and multifault testable*. For control portions of VLSI circuits, they use algebraic factorization procedures described above that guarantee path-delay-fault or multifault testability, starting from a sum-of-products representation of a function. They use *hierarchical composition rules* [70] in the synthesis of regular structures occurring in datapath portions, like parity generators and arithmetic units. Test vectors to detect all path delay faults and multifaults can be obtained as a by-product of the synthesis process. They have successfully used these techniques on circuits with over 5000 gates. They present preliminary experimental results on a data encryption chip implementing the Viterbi algorithm, a small μ-processor and a speech recognition chip [28].

In the area of implementation verification, they have developed new, efficient algorithms for sequential logic verification based on the notion of implicit state enumeration [10]. These algorithms achieve significant speed-ups over previous approaches to sequential logic verification.

Verifying that a logic-level description correctly implements a behavioral specification is considerably less developed. One major hindrance toward a precise notion of behavioral verification has been that parallel, serial or pipelined implementations of the same behavioral description can be implemented in finite-state automata with different input/output behaviors. They have used *nondeterminism* to model the degree of freedom that is afforded by parallelism in a behavioral description that also contains complex control. Given some assumptions, the set of finite automata derivable from a behavioral description under all possible schedules of operations can be represented compactly as an *input-programmed automaton* (p-Automaton). The p-Automaton is named such due to the fact that during its derivation, they program *meta-input variables* in the p-Automaton that are not present in the original description. The logic-level implementation is deemed to be equivalent to the behavioral description if and only if the p-Automaton is equivalent to the logic-level finite automaton *under some assignment to the meta-input variables*. The above method allows for extending the use of finite-state automata equivalence-checking algorithms to the problem of behavioral verification [27].

In the area of synthesis for sequential logic testability, Devadas' group has shown how register-transfer-level (RTL) descriptions of a circuit can be used to efficiently synthesize circuits with over 1000 latches for full non-scan single stuck-at fault testability. They have synthesized a version of the Viterbi processor for full testability with no area or performance overhead [39]. Control portions of VLSI circuits can be synthesized for sequential testability using recently developed decomposition algorithms [8]. A problem occurring in the synthesis of controllers for non-scan testability requires optimization of *Boolean relations*. They have developed heuristic procedures that optimize for Boolean relations guaranteeing a locally optimal solution [38].

Over the past six months, Jacob White's efforts in developing numerical algorithms for problems related to the design of an ARC, as well as those that can effectively exploit the ARC's capability, have yielded several interesting results. His group now has useful programs for capacitance calculation and hydrodynamic device simulation, as well as parallel implementions of both a device simulator and a specialized circuit simulator. The parallel device simulator achieves a 15 times speed-up on a 16 processor INTEL hypercube, and the parallel circuit simulator exploits the Connection Machine effectively enough to achieve a 1400 times speed-up over a SUN4 workstation. Below are described these results, and few others, in more detail.

Three dimensional capacitance and inductance extraction has recently become important because the dense packing of processors and memory required for high performance parallel computers require three dimensional interconnection. White's group has finished developing a fast algorithm for computing the capacitance of a complicated 3-D geometry of ideal conductors in a uniform dielectric. The method is an acceleration of the standard integral equation approach for multiconductor capacitance extraction. These integral equation methods are slow because they lead to dense matrix problems which are typically solved with some form of Gaussian elimination. This implies the computation grows like n^3, where n is the number of tiles needed to accurately discretize the conductor surface charges. We have developed a preconditioned conjugate-gradient iterative algorithm with a multipole approximation to compute the iterates. This reduces the complexity of the multiconductor capacitance calculations to grow roughly as nm where m is the number of conductors.

Performance comparisons on integrated circuit bus crossing problems show that for problems with as few as twelve conductors, the multipole accelerated boundary element method in their program, FASTCAP,

can be as much as 500 times faster than more classic Gaussian elimination based boundary-element algorithms, and five to ten times faster than the iterative method alone, depending on required accuracy [125]. They recently generalized the program to interface with the solid-modeling program PATRAN, used by the MIT CAF project. The generalization of FASTCAP, and the interfacing involved, provides three new features: an excellent user interface; the ability to compute the capacitance of any general 3-D structure, including curved objects like spheres; and the MIT CAF project can immediately use the program to analyze electrostatic properties of microstructures.

In the area of circuit simulation, a specialized parallel simulator for grid-based analog signal processing arrays has been implemented on the massively parallel Connection Machine. Standard implicit integration techniques are used in the program, except that a parallel block conjugate-gradient squared algorithm is used to solve the linear systems generated for each integration timestep. Excellent parallel performance of the algorithm is achieved through the use of a novel, but very natural, mapping of the circuit data onto the massively parallel architecture. The mapping takes advantage of the underlying connection machine architecture and the structure of the analog array problem. Experimental results using their program, CMVSIM [35], demonstrate that a full-size Connection Machine running their parallel algorithm can simulate a realistic analog array 1400 times faster a SUN4/280 workstation running the fastest of the known serial algorithms.

Also in the area of circuit simulation, White's group has completed some new work on methods for simulating clocked analog circuits like switching power converters, switched-capacitor filters, and phase-locked loops. Simulating these circuits is computationally expensive because they are clocked at a frequency whose period is orders of magnitude smaller than the time interval of interest to the designer. It is possible to reduce the simulation time without compromising much accuracy by exploiting the property that the behavior of switching converters in a given high-frequency clock cycle is similar, but not identical, to the behavior in the preceding and following cycles. In particular, the "envelope" of the high-frequency clock can be followed by accurately computing the circuit behavior over occasional cycles. They implemented such a method in the program NITSWIT, but the method was only effective on simplified circuits, and was *not* able to skip many cycles on realistic circuits [58]. The difficulty was that realistic circuits typically include circuitry which produces large rapid responses to small changes in slow moving variables. These rapidly responding variables must be somehow eliminated before introducing the envelope-following, otherwise the cycle skipping is severely limited. They are investigating automatic ways of eliminating those variables based on examining the sensitivity matrix.

In the area of device simulation, White's group is continuing their work on parallel 2-D MOS device simulation. They have a prototype simulator, which does steady-state calculations, running on the INTEL hypercube. The program finishes 15 times faster on 16 processors than on one processer. They are currently working on parallelizing their waveform relaxation algorithm for the transient calculations [64], as well as improving the physical models in the simulator.

Although still useful for predicting terminal currents, the drift-diffusion model of electron transport does not include enough information to accurately predict the carrier energy distribution in small geometry devices. This is of particular importance for predicting oxide breakdown due to penetration by "hot" electrons. They have been working on the numerical algorithms for solving the "hydrodynamic equations," for electron transport, in which an energy balance equation is solved along with the drift-diffusion equations. They have uncovered a discretization method for the hydrodynamic problem which is much more stable than other approaches, and allows very coarse meshes to be used during the simulation.

8 Publications

8.1 Journal and Conference Publications

[1] Anant Agarwal and Minor Huffman. Blocking: Exploiting Spatial Locality for Trace Compaction. In *Proceedings of ACM SIGMETRICS 1990*, May 1990.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[3] A. Aggarwal, D. Kravets, J. Park, and S. Sen. Parallel searching in generalized Monge arrays with applications. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 259–268, July 1990.

[4] A. Aggarwal, T. Leighton, and K. Palem. Area-time optimal circuits for iterated addition in VLSI. *IEEE Transactions on Computers*, 1990. To appear.

[5] A. Aggarwal and J. Park. Improved algorithms for economic lot-size problems. Research Report RC 15626, IBM Research Division, T. J. Watson Research Center, March 1990. Submitted to *Operations Research*.

[6] W. Aiello, T. Leighton, B. Maggs, and M. Newman. Fast algorithms for bit-serial routing on a hypercube. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 55–64, 1990.

[7] Sanjeev Arora, Tom Leighton, and Bruce Maggs. On-line algorithms for path selection in a nonblocking network. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 149–158, 1990.

[8] P. Ashar, S. Devadas, and A. R. Newton. Testability-Driven Decomposition of Large Finite State Machines. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, September 1990.

[9] P. Ashar, S. Devadas, and A. R. Newton. A unified approach to decomposition and re-decomposition of sequential machines. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[10] P. Ashar, A. Ghosh, S. Devadas, and A. R. Newton. Implicit State Transition Graphs: Applications to Sequential Logic Synthesis and Test. In *Proceedings of the Int'l Conference on Computer-Aided Design*, November 1990. To appear.

[11] B. Berger, M. Brady, D. Brown, and T. Leighton. Nearly optimal algorithms and bounds for multilayer channel routing. *J. ACM*, 1990. To appear.

[12] F. Berman, D. Johnson, T. Leighton, P. Shor, and L. Snyder. Generalized planar matching. *J. Alg.*, 11(2):153–184, 1990.

[13] S. Bhatt, F. Chung, J. Hong, T. Leighton, and A. Rosenbery. Optimal simulations of butterfly networks. *J. ACM*, 1990. To appear.

[14] S. Bhatt, F. Chung, T. Leighton, and A. Rosenbery. Efficient embeddings of trees in hypercubes. *SIAM Journal on Computing*, 1990. To appear.

[15] M. J. Bryan, S. Devadas, and K. Keutzer. Testability-Preserving Circuit Transformations. In *Proceedings of the Int'l Conference on Computer-Aided Design*, November 1990. To appear.

[16] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[17] David L. Chaiken. *Cache Coherence Protocols for Large Scale Multiprocessors.* MIT Master's thesis, September 1990.

[18] Andrew Chien and William J. Dally. Concurrent aggregates (ca). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–196. ACM SIGPLAN Notices 25:3, March 1990.

[19] Andrew A. Chien. *Concurrent Aggregates: An Object-Oriented Language for Fine-Grained Message-Passing Machines.* PhD thesis, Massachusetts Institute of Technology, May 1990.

[20] Andrew A. Chien and William J. Dally. Experience with concurrent aggregates (ca): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*, Charleston, South Carolina, April 8-12 1990. SIAM.

[21] William J. Dally. A fast translation method for paging on top of segmentation. *IEEE Transactions on Computers*, 1989. Accepted for publication. Also appears as VLSI Memo 89-502.

[22] William J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 1990. Accepted as a Regular Paper.

[23] William J. Dally. Network and processor architecture for message-driven computers. In *VLSI and Parallel Computation*, pages 140–222. Kluwer Academic Publishers, 1990.

[24] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6), June 1990. Also appears as a chapter in *Artificial Intelligence at MIT, Expanding Frontiers*, pp. 548-581.

[25] William J. Dally. Virtual-channel flow control. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 60–68, Los Alamitos, CA, May 1990. IEEE Computer Society Press.

[26] S. Devadas. Minimization of functions with multiple-valued outputs: Theory and applications. In *Proceedings of the 20th Int'l Symposium on Multiple-Valued Logic*, May 1990.

[27] S. Devadas and K. Keutzer. An Automata-Theoretic Approach to Behavioral Equivalence. In *Proceedings of the Int'l Conference on Computer-Aided Design*, November 1990. To appear.

[28] S. Devadas and K. Keutzer. Design of Integrated Circuits Fully Testable for Delay-Faults and Multifaults. In *Proceedings of the Int'l Test Conference*, September 1990.

[29] S. Devadas and K. Keutzer. Necessary and sufficient conditions for robust delay-fault testability of combinational logic circuits. In W.J. Dally, editor, *Proceedings of the Sixth MIT Conference On Advanced Research in VLSI*, pages 221–238, Cambridge, Massachusetts, April 2–4 1990.

[30] S. Devadas and K. Keutzer. Synthesis and optimization procedures for robust delay-fault testability of logic circuits. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[31] S. Devadas and K. Keutzer. Synthesis for testability: A brief survey. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990.

[32] S. Devadas and K. Keutzer. Validatable nonrobust delay-fault testable circuits via logic synthesis. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990.

[33] S. Devadas and K. Keutzer. A unified approach to the synthesis of fully testable sequential machines. In *IEEE Transactions on Computer-Aided Design*, January 1991. To appear.

[34] S. Devadas, K. Keutzer, and J. White. Estimation of power dissipation in CMOS combinational circuits. In *Proceedings of the Custom Integrated Circuits Conference*, May 1990.

[35] S. Devadas, K. Keutzer, and J. White. Estimation of power dissipation in CMOS combinational circuits. In *Proc. Custom Int. Circuits Conf.*, Boston, May 1990.

[36] S. Devadas and A. R. Newton. Exact Algorithms for Output Encoding, State Assignment and Four-Level Boolean Minimization. In *IEEE Transactions on Computer-Aided Design*, January 1991. To appear.

[37] M. Formann, T. Hagerup, J. Haralambides, T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger. Drawing graphs in the plane with high resolution. In 31st *IEEE Symposium on Theory of Computing*, October 1990. To appear.

[38] A. Ghosh, S. Devadas, and A. R. Newton. Heuristic Minimization of Boolean Relations Using Testing Techniques. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, September 1990.

[39] A. Ghosh, S. Devadas, and A. R. Newton. Sequential Logic Synthesis for Testability Using Register-Transfer-Level Descriptions. In *Proceedings of the Int'l Test Conference*, September 1990.

[40] A. Ghosh, S. Devadas, and A. R. Newton. Sequential test generation at the register-transfer and logic levels. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[41] A. Ghosh, S. Devadas, and A. R. Newton. Verification of interacting sequential circuits. In *Proceedings of the 27th Design Automation Conference*, June 1990.

[42] A. Ghosh, S. Devadas, and A. R. Newton. Test Generation and Verification of Highly Sequential Circuits. In *IEEE Transactions on Computer-Aided Design*, January 1991.

[43] Alexander T. Ishii. A timing verification algorithm for level-clocked circuitry. In W.J. Dally, editor, *The Sixth MIT Conference on Advanced Research in VLSI*, pages 113–130, Cambridge, Massachusetts, April 2-4, 1990.

[44] C. Kaklamanis, A. Karlin, T. Leighton, V. Milenkovic, G. Nelson, P. Raghavan, S. Rao, C. Thomborson, and T. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In 31^{st} *IEEE Symposium on Theory of Computing*, October 1990. To appear.

[45] Joe Kilian, Shlomo Kipnis, and Charles E. Leiserson. The organization of permutation architectures with bussed interconnections. *IEEE Transactions on Computers*, 39:00–00, 1990. To appear. Also appeared as technical memo MIT/LCS/TM-379 and VLSI memo 89-500. Earlier version appeared in 28^{th} IEEE Annual Symposium on Foundations of Computer Science (1987), 305–315.

[46] Shlomo Kipnis. Priority arbitration with busses. In W.J. Dally, editor, *the Sixth MIT Conference on Advanced Research in VLSI*, Cambridge, Massachusetts, April 2–4, 1990. Also appears as MITLCS TM-408, October, 1989.

[47] M. Klawe and T. Leighton. A tight lower bound on the size of planar permutation layouts. In *Proc. SIGAL Int. Symp. on Algorithms*, Japan, August 1990.

[48] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp (Extended Abstract). In *Parallel Lisp: Languages and Systems, (Springer-Verlag Lecture Notes in Computer Science 441)*, 1990.

[49] K. Kundert, J. White, and A. Sangiovanni-Vincentelli. *Steady-State Methods for Simulating Analog and Microwave Circuits*. Kluwer Academic Press, Boston, 1990.

[50] K. Lam and S. Devadas. Performance-oriented synthesis of finite state machines. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990.

[51] Kevin Lam, Larry Dennison, and William Dally. Simultaneous bidirectional signaling for IC systems. In *ICCD '90*. IEEE Computer Society, September 1990.

[52] T. Leighton. A $2d-1$ lower bound for ? ayer knock-knee channel routing. *Siam Journal of Discrete Mathematics*, 1990. To appear.

[53] T. Leighton. Average case analysis of greedy routing algorithms on arrays. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 2-10, July 1990.

[54] T. Leighton, D. Lisinski, and B. Maggs. Empirical evaluation of randomly-wired multistage networks. In *Proceedings of the 1990 International Conference on Computer Design*. IEEE, September 1990.

[55] T. Leighton, F. Makedone, and D. Bhatia. Efficient reconfiguration on WSI arrays. In *Proc. 1^{st} ACM-IEEE Int. Conf. on System Integration*, pages 46–57, April 1990.

[56] T. Leighton, F. Makedone, and S. Tragoudas. Approximation algorithms for VLSI partition problems. In *Proc. 1^{st} ACM-IEEE Int. Conf. on Circuits and Systems*, pages 2865–2869, May 1990.

[57] T. Leighton and C. G. Plaxton. A (fairly) simple circuit that (usually) sorts. In 31st *IEEE Symposium on Theory of Computing*, October 1990. To appear.

[58] J. Lloyd, J. Phillips, and J. White. A boundary-element/multipole algorithm for self-consistent field calculations in monte-carlo simulation. In *Proc. Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits: NUPAD III*, Honolulu, Hawaii, June 1990.

[59] F. Makedone, T. Leighton, and I. Tollis. A $2N - 2$ step algorithm for routing on an $N \times N$ array with constant size queues. *J. Algorithmica*, 1990. To appear.

[60] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel tasks. In *Proceedings of Symposium on Lisp and Functional Programming*, June 1990.

[61] Michael Noakes and William J. Dally. System design of the j-machine. In William J. Dally, editor, *Proceedings of the Sixth MIT Conference of Advanced Research in VLSI*, pages 179–194, Cambridge, MA 02139, April 2-4 1990. The MIT Press.

[62] Dan Nussbaum, Ingmar Vuong, and Anant Agarwal. Modeling a Circuit-Switched Multiprocessor Interconnect. In *Proceedings of ACM SIGMETRICS 1990*, May 1990.

[63] S. Devadas P. Ashar and A. R. Newton. Multiple-fault testable sequential machines. In *Proceedings of the Int'l Conference on Circuits and Systems*, May 1990.

[64] R. Saleh and J. White. Accelerating relaxation algorithms for circuit simulation using waveform-newton and step-size refinement. In *IEEE Transactions on Computer-Aided Design*, October 1990.

[65] Ellen Spertus. Dataflow computation on the J-Machine. Bachelor's Thesis, May 1990.

8.2 Internal Memoranda

[66] Anant Agarwal. A Locality-Based Multiprocessor Cache Interference Model. Submitted for publication.

[67] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. Submitted for publication to *IEEE Transactions on Parallel and Distributed Systems*. Also appears as MIT VLSI Memo 89-566.

[68] Anant Agarwal. Limits on Network Performance. To appear as a MIT VLSI Memo. Also submitted for publication to *IEEE Transactions on Parallel and Distributed Systems*, November 1989.

[69] A. Aggarwal and J. Park. Algorithms for economic lot-size problems with bounds on inventory and backlogged demand. Manuscript in preparation, 1990.

[70] M. J. Bryan, S. Devadas, and K. Keutzer. Analysis and design of regular structures for robust dynamic fault testability. Submitted to the Int'l Symposium on Circuits and Systems, 1990.

[71] Waldemar Horwat, Brian Totty, and William J. Dally. Cosmos: An operating system for a fine-grain concurrent computer. Submitted for publication, 1990.

[72] Shlomo Kipnis. *Organization of Systems with Bussed Interconnections.* PhD thesis, MIT Lab. for Computer Science, August 1990.

[73] David A. Kranz, David Chaiken, and Anant Agarwal. Multiprocessor Address Tracing and Performance Analysis. Submitted to ACM SIGMETRICS 1991. Also to appear as a MIT VLSI Memo, September 1990.

[74] T. Leighton, D. Kravets, and C. Leiserson. Lecture notes for 18.435/6.848 Theory of Parallel and VLSI Computation, fall, 1989. Technical Report MIT/LCS/RSS 8, MIT LCS Research Seminar Series, May 1990.

[75] T. Leighton, B. Maggs, A. Ranade, and S. Rao. Randomized algorithms for routing and sorting in fixed-connection networks. Submitted to *Journal of Algorithms*.

[76] K. Nabors and J. White. Fastcap: A multipole-accelerated 3-d capacitance extraction program. Submitted to *the IEEE Trans. on Comp. Aided Design*, 1990.

[77] Dan Nussbaum and Anant Agarwal. Scalability of Parallel Machines. To appear as a MIT VLSI Memo, July 1990.

[78] Peter Nuth and William J. Dally. The named state processor. To appear as a MIT VLSI Memo, MIT, 1990.

[79] Peter R. Nuth. Parallel processor architecture: A thesis proposal. MIT VLSI Memo, 1990.

[80] Marios Christos C. Papaefthymiou. On retiming synchronous circuitry and mixed-integer optimization. Technical Report TR-486, MIT Lab. for Computer Science, to appear in October 1990. Also to appear as a VLSI Memo.

[81] L. Silveira, J. White, H. Neto, and L. Vidigal. On exponential fitting for circuit simulation (expanded). Submitted to the IEEE Transactions on Computer-Aided Design, 1990.

[82] Ellen Spertus and William Dally. Dataflow on a general-purpose parallel computer. Submitted for publication, 1990.

[83] J. White and S. Leeb. An envelope-following approach to switching power converter simulation. Submitted to the IEEE Transactions on Power Electronics.

8.3 Talks without Proceedings

[84] Anant Agarwal. April: A processor architecture for multiprocessing. Lecture given at 17th Annual Symposium on Computer Architecture, Seattle, June *1990*.

[85] Anant Agarwal. Blocking: Exploiting spatial locality for trace compaction. Lecture given at ACM SIGMETRICS Conference, Boulder, May 1990.

[86] Anant Agarwal. Exploiting locality for scalability. Lecture given at IEEE Workshop on Interconnections within Digital Systems, Santa Fe, May 1990.

[87] Anant Agarwal. The MIT ALEWIFE machine: A scalable cache-coherent multiprocessor. Lecture given at Large Systems Seminar, IBM T. J. Watson Research Center, September 1990.

[88] Anant Agarwal. The MIT ALEWIFE machine: A scalable cache-coherent multiprocessor. Lecture given at JPL, Pasadena; SUN Microsystems, Mountain View, California; LSI Logic, California, August 1990.

[89] Anant Agarwal. The MIT ALEWIFE machine: Exploiting locality for scalability. Lecture given at Distinguished Lecture Series. Encore Computer Corp., April 1990.

[90] Anant Agarwal. The MIT ALEWIFE machine: Exploiting locality for scalability. Lecture given at Siemens, Munich, July 1990.

[91] Anant Agarwal. Multiprocessor address tracing: The agony and the ecstasy. Lecture given at Workshop on Multiprocessor Performance Evaluation, Seattle, May 1990.

[92] Anant Agarwal. The numesh: A scalable modular 3-d interconnect. Lecture given at Siemens, Munich, July 1990.

[93] Anant Agarwal. Processor architecture for multiprocessing. Lecture given at Workshop on Multithreaded Processors, Laboratory for Computer Science, MIT, January 1990.

[94] Anant Agarwal and Beng-Hong Lim. Exploiting locality for scalability in the mit alewife machine. Lecture given at Workshop on Shared Memory Multiprocessors. Seattle, May 1990.

[95] W.J. Dally. J-machine. Lecture given at Intel Corporation, Santa Clara, California, Digital Equipment Corporation in Palo Alto, California, May 1990.

[96] W.J. Dally. Parallel computer architecture, parallel computer software, and parallel computer networks. Lecture given at Intel Corporation, Santa Clara, California, August 1990.

[97] W.J. Dally. Simultaneous bidirectional signaling for ic systems. Lecture given at ICCD '90, Cambridge, Massachusetts, September 1990.

[98] W.J. Dally. System design of the j-machine. Lecture given at Sixth MIT Conference of Advanced Research in VLSI, Cambridge, Massachusetts, April 1990.

[99] W.J. Dally. Virtual flow control. Lecture given at the 17th Annual International Symposium on Computer Architecture, Seattle, Washington, May 1990.

[100] S. Devadas. Design of integrated circuits fully testable for delay-faults and multifaults. Lecture given at Int'l Test Conference, Washington D.C., September 11, 1990.

[101] S. Devadas. Panel discussion on "testing strategies for the 1990s". Lecture given at 27^{th} Design Automation Conference, Orlando, June 26, 1990.

[102] S. Devadas. Tutorial on synthesis of sequential circuits. Lecture given at 27^{th} Design Automation Conference, Orlando, June 28, 1990.

[103] S. Devadas. Validatable nonrobust delay-fault testable circuits via logic synthesis. Lecture given at Int'l Symposium on Circuits and Systems, New Orleans, May 1990.

[104] Shlomo Kipnis. Organization of systems with bussed interconnections. Lecture given at IBM Almaden Research Center, IBM Yorktown Research Center, Philips Laboratories, Digital Systems Research Center, AT&T Bell Laboratories. Columbia University, MIT, 1990.

[105] T. Knight. Advanced architectures for artificial intelligence applications. Lecture given at Artificial Intelligence Conference, Rabat, Morocco, 1990.

[106] T. Knight. Comparison of routing protocols. Lecture given at MIT Workshop on Large Scale VLSI Systems, June 1990.

[107] T. Knight. High speed video interconnection switching technologies. Lecture given at Grass Valley Group, Inc., August 1990.

[108] T. Knight. The limits of electronic interconnection technologies. Lecture given at IEEE Workshop on Optical and Electronic Interconnect, Santa Fe, NM, May 1990.

[109] T. Knight. The transit interconnection architecuture and packaging. Lecture given at Sun Microsystems, June 1990.

[110] David Kranz. Parallel lisp: Present and future. Lecture given at INRIA Rocquencourt, France, July 2, 1990.

[111] John Kubiatowicz. Hardware-software tradeoffs in the alewife multiprocessor. Lecture given at Workshop on Shared Memory Multiprocessors, Seattle, May 1990.

[112] Kevin Lam, Larry Dennison, and William Dally. Simultaneous bidirectional signaling for IC systems. Lecture given at ICCD '90, September 1990.

[113] Tom Leighton. Average-case analysis of greedy routing algorithms on arrays. Lecture given at ACM Symposium on Parallel Algorithms and Architectures, Greece, July 1990.

[114] Tom Leighton. Fast fault-tolerant algorithms for routing on multibutterflies and nonblocking networks. Lecture given at Distinguished lecture at MIT Conference on Advanced Research in VLSI; Thinking Machines, Cambridge; Distinguished Lecture at the Workshop on Parallel Algorithms, Annapolis; Distinguished Lecuture at the ARIDAM Workshop, Rutgers, 1990.

[115] Tom Leighton. The role of randomness in the design of parallel architectures. Lecture given at Distinguished lecture at MIT Conference on Advanced Research in VLSI, April 1990.

[116] Beng-Hong Lim. A run-time system for the alewife machine. Lecture given at Systems Research Center, Digital Equipment Corporation; TOPAZ Users group meeting, June 1990.

[117] Bruce Maggs. Asymptotically optimal schedules for packet routing. Lecture given at SIAM Annual Meeting, Chicago Illinois, July 20, 1990.

[118] Bruce Maggs. Fault-tolerant routing algorithms for multibutterfly networks. Lecture given at NEC Research Institute, Bell Communications Research Center, Rice University, IBM T. J. Watson Research Center, Duke University, ATT Bell Laboratories, Cornell University, Harvard University, 1990.

[119] J. White. A boundary-element/multipole algorithm for self-consistent field calculations in monte-carlo simulation. Lecture given at Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits: NUPAD III, Honolulu, Hawaii, June 4, 1990.

[120] J. White. Estimation of power dissipation in cmos combinational circuits. Lecture given at Custom Int. Circuits Conf., Boston, May 15, 1990.

[121] J. White. Numerical techniques for integrated circuit design problems. Lecture given at Bell Laboratories, Allentown, PA, May 8, 1990.

[122] J. White. On exponential fitting for circuit simulation. Lecture given at Proc. Int. Symp. on Circuits and Systems, New Orleans, May 2, 1990.

8.4 Selected Publications

[123] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: a scalable cache coherence scheme. To appear as a MIT VLSI Memo. Also submitted for publication, August 1990.

[124] Bill Dally and Hiromichi Aoki. Adaptive routing using virtual channels. To appear as a MIT VLSI Memo. Also submitted for publication, September 1990.

[125] J. White L. M. Silveira, A. Lumsdaine. Parallel simulation algorithms for grid-based analog signal processors. In *Proc. Int. Conf. on Computer-Aided Design*, October 1990. To appear.

LimitLESS Directories: A Scalable Cache Coherence Scheme*

David Chaiken, John Kubiatowicz, and Anant Agarwal
Laboratory for Computer Science, NE43–633
Massachusetts Institute of Technology
Cambridge, MA 02139
(617) 253 – 1448
chaiken@vindaloo.lcs.mit.edu

Abstract

Caches enhance the performance of multiprocessors by reducing network traffic and average memory access latency. However, cache-based systems must address the problem of cache coherence. We propose the LimitLESS directory protocol to solve this problem. The LimitLESS scheme uses a combination of hardware and software techniques to realize the performance of a full-map directory with the memory overhead of a limited directory. This protocol is supported by Alewife, a large-scale multiprocessor. We describe the architectural interfaces needed to implement the LimitLESS directory, and evaluate its performance through simulations of the Alewife machine.

1 Introduction

The communication bandwidth of interconnection networks is a critical resource in large-scale multiprocessors. This situation will remain unchanged in the future because physically constrained communication speeds cannot match the increasing bandwidth requirements of processors that leverage off of rapidly advancing VLSI technology. Caches reduce the volume of traffic imposed on the network by automatically replicating data where it is needed. When a processor attempts to read or to write a unit of data, the system fetches the data from a remote memory module into a cache, which is a fast local memory dedicated to the processor. Subsequent accesses to the same data are satisfied within the local processing node, thereby avoiding repeat requests over the interconnection network.

In satisfying most memory requests, a cache increases the performance of the system in two ways: First, memory access latency incurred by the processors is shorter than in a system that does not cache data, because typical cache access times are much lower than interprocessor communication times (often, by several orders of magnitude). Second, when most requests are satisfied within processing nodes, the volume of network traffic is also lower.

However, replicating blocks of data in multiple caches introduces the cache coherence problem. When multiple processors maintain cached copies of a shared memory location, local

*Submitted to ASPLOS-IV, 1991.

modifications can result in a globally inconsistent view of memory. Buses in small-scale multiprocessors offer convenient solutions to the coherence problem that rely on system-wide broadcast mechanisms [1, 2, 3, 4, 5]. When any change is made to a data location, a broadcast is sent so that all of the caches in the system can either invalidate or update their local copy of the location. Unfortunately, this type of broadcast in large-scale multiprocessors negates the bandwidth reduction that makes caches attractive in the first place. Furthermore, in large-scale multiprocessors, broadcast mechanisms are either inefficient or prohibitively expensive to implement.

A number of cache coherence protocols have been proposed to solve the coherence problem in the absence of broadcast mechanisms [6, 7, 8, 9]. These message-based protocols allocate a section of the system's memory, called a directory, to store the locations and state of the cached copies of each data block. Instead of broadcasting a modified location, the memory system sends an invalidate (or update) message to each cache that has a copy of the data. The protocol must also record the acknowledgment of each of these messages to ensure that the global view of memory is actually consistent.

Although directory protocols have been around since the late seventies, the usefulness of the early protocols (e.g., [7]) was in doubt for several reasons: First, the directory itself was a *centralized* monolithic resource which serialized all requests. Second, directory accesses were expected to consume a disproportionately large fraction of the available network bandwidth. Third, the directory became prohibitively large as the number of processors increased. To store pointers to blocks potentially cached by all the processors in the system, the early directory protocols (such as the Censier and Feautrier scheme [7]) allocate directory memory proportional to the product of the total memory size and the number of processors. While such *full-map* schemes permit unlimited caching, its directory size grows as $O(N^2)$, where N is the number of processors in the system.

As observed in [8], the first two concerns are easily dispelled: The directory can be *distributed* along with main memory among the processing nodes to match the aggregate bandwidth of distributed main memory. Furthermore, required directory bandwidth is not much more than the memory bandwidth, because accesses destined to the directory alone comprise a small fraction of all network requests. Thus, recent research in scalable directory protocols focuses on alleviating the severe memory requirements of the distributed full-map directory schemes.

Scalable coherence protocols differ in the size and the structure of the directory memory that is used to store the locations of cached blocks of data. *Limited directory* protocols [8], for example, avoid the severe memory overhead of full-map directories by allowing only a limited number of simultaneously cached copies of any individual block of data. Unlike a full-map directory, the size of a limited directory grows linearly with the size of shared memory, because it allocates only a small, fixed number of pointers per entry. Once all of the pointers in a directory entry are filled, the protocol must evict previously cached copies to satisfy new requests to read the data associated with the entry. In such systems, widely shared data locations degrade system performance by causing constant eviction and reassignment, or *thrashing*, of directory pointers. However, previous studies have shown that a small set of pointers is sufficient to capture the *worker-set* of processors that concurrently read many types of data [10, 11, 12]. The performance of limited directory schemes can approach the performance of full-map schemes if the software is optimized to minimize the number of widely-shared objects.

This paper proposes the LimitLESS cache coherence protocol, which realizes the performance

of the full-map directory protocol, with the memory overhead of a limited directory, but without excessive sensitivity to software optimization. This new protocol is supported by the architecture of the Alewife machine, a large-scale, distributed-memory multiprocessor. Each processing node in the Alewife machine contains a processor, a floating-point unit, a cache, and portions of the system's globally shared memory and directory. The LimitLESS scheme implements a small set of pointers in the memory modules, as do limited directory protocols. But when necessary, the scheme allows a memory module to interrupt the processor for software emulation of a full-map directory. Since this new coherence scheme is partially implemented in software, it can work closely with a multiprocessor's compiler and run-time system.

Chained directory protocols [9], another scalable alternative for cache coherence, avoid both the memory overhead of the full-map scheme and the thrashing problem of limited directories by distributing directory pointer information among the caches in the form of linked lists. But unlike the LimitLESS scheme, chained directories are forced to transmit invalidations sequentially through a linked-list structure, and thus incur high write latencies for very large machines. Furthermore, the chained directory protocol lacks the LimitLESS protocol's ability to couple closely with a multiprocessor's software, as described in Section 6.

To evaluate the LimitLESS protocol, we have implemented the full-map directory, limited directory, and other cache coherence protocols in ASIM, the Alewife system simulator. Since ASIM is capable of simulating the entire Alewife machine, the different coherence schemes can be compared in terms of absolute execution time. While we have used more generic metrics (such as processor utilization or cycles per transaction) in past studies [10], simulated execution time gives the closest approximation of the behavior of an actual multiprocessing system.

The next section describes the details of the Alewife machine's architecture that are relevant to the LimitLESS directory protocol. Section 3 introduces the LimitLESS protocol, and Section 4 presents the architectural interfaces and various hardware and software mechanisms needed to implement the new coherence scheme. Section 5 describes the Alewife system simulator and compares the different coherence schemes in terms of absolute execution time. Section 6 suggests extensions to the software component of the LimitLESS scheme that couple the coherence protocol with the machine's runtime system, and Section 7 summarizes the results and discusses future work in this area.

2 The Alewife Machine

Alewife is a large-scale multiprocessor with distributed shared memory. The machine, organized as shown in Figure 1, uses a cost-effective mesh network for communication. This type of architecture scales in terms of hardware cost and allows the exploitation of locality. Unfortunately, the non-uniform communication latencies make such machines hard to program because the onus of managing locality invariably falls on the programmer. The goal of the Alewife project is to discover and to evaluate techniques for automatic locality management in scalable multiprocessors in order to insulate the programmer from the underlying machine details. Our approach to achieving this goal employs techniques for *latency minimization* and *latency tolerance*.

Several mechanisms in the Alewife compiler, runtime system, and hardware cooperate in enhancing communication locality, thereby reducing communication latency and required network bandwidth. Shared-data caching in Alewife is an example of a hardware method for reduc-

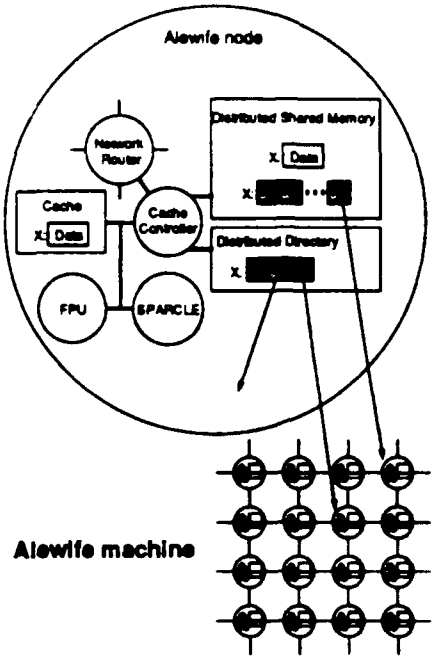

Figure 1: An Alewife processing node with a LimitLESS directory extension.

ing communication traffic. This method is dynamic (uses run-time information), rather that static (compiler-specified). Lazy task creation [13] together with near-neighbor scheduling are Alewife's software methods for achieving the same effect.

When the system cannot avoid a remote memory request and is forced to incur the latency of the communication network, the Alewife processors rapidly schedule another process in place of the stalled process. Alewife can also tolerate synchronization latencies through the same context switching mechanism. Because context switches are forced only on memory requests that require the use of the interconnection network, and on synchronization faults, the processor achieves high single-thread performance. Some systems [14] have opted to use weak ordering [15, 16, 17] to tolerate certain types of communication latency, but this method lacks the ability to overlap read-miss and synchronization latencies. Although the Alewife cache coherence protocol enforces sequential consistency [18], the LimitLESS directory scheme can also be used with a weakly-ordered memory model.

We have designed a new processor architecture that can rapidly switch between processes [19]. The first-round implementation of the processor called SPARCLE will switch between processes in 11 cycles. This fast context-switch is achieved by caching four sets of register frames on the processor to eliminate the overhead of loading and unloading the process registers. The rapid-switching features of SPARCLE also allow an efficient implementation of LimitLESS directories.

An Alewife node consists of a 33 MHz SPARCLE processor, 64K bytes of direct-mapped cache, a 4M bytes of globally-shared main memory, and a floating-point coprocessor. Both the cache and floating-point units are SPARC compatible [20]. The nodes communicate via messages through a direct network [21] with a mesh topology using wormhole routing [22]. A single-chip controller on each node holds the cache tags and implements the cache coherence

protocol by synthesizing messages to other nodes. Figure 1 is an enlarged view of a node in the Alewife machine. Because the directory itself is distributed along with the main memory, its bandwidth scales with the number of processors in the system. The SPARCLE processor is being implemented jointly with LSI Logic and SUN Microsystems through modifications to an existing SPARC design. The design of the cache/memory controller is also in progress.

3 The LimitLESS Directory Protocol

As do limited directory protocols, the LimitLESS directory scheme capitalizes on the observation that only a few shared memory data types are widely shared among processors. Many shared data structures have a small *worker-set*, which is defined as the set of processors that concurrently read a memory location. The worker-set of a memory block corresponds to the number of active pointers it would have in a full-map directory entry. The observation that worker-sets are often small has led some memory-system designers to propose the use of a hardware cache of pointers to augment the limited-directory for a few widely-shared memory blocks [12]. However, when running properly optimized software, a directory entry overflow is an exceptional condition in the memory system. We propose to handle such "protocol exceptions" in software. This is the integrated systems approach — handling common cases in hardware and exceptional cases in software.

The LimitLESS scheme implements a small number of hardware pointers for each directory entry. If these pointers are not sufficient to store the locations of all of the cached copies of a given block of memory, then the memory module will interrupt the local processor. The processor will then emulate a full-map directory for the block of memory that caused the interrupt. The structure of the Alewife machine provides for an efficient implementation of this memory system extension. Since each processing node in Alewife contains both a memory controller and a processor, it is a straightforward modification of the architecture to couple the responsibilities of these two functional units. This scheme is called LimitLESS, to indicate that it employs a *Limit*ed directory that is *L*ocally *E*xtended through *S*oftware *S*upport. Figure 1 is an enlarged view of a node in the Alewife machine. The diagram depicts a set of directory pointers that correspond to the shared data block X, copies of which exist in several caches. In the figure, the software has extended the directory pointer array (which is shaded) into local memory.

Since Alewife's SPARCLE processor is designed with a fast trap mechanism, the overhead of the LimitLESS interrupt is not prohibitive. The emulation of a full-map directory in software prevents the LimitLESS protocol from exhibiting the sensitivity to software optimization that is exhibited by limited directory schemes. But given current technology, the delay needed to emulate a full-map directory completely in software is significant. Consequently, the LimitLESS protocol supports small worker-sets of processors in its limited directory entries, implemented in hardware.

3.1 A Simple Model of the Protocol

Before discussing the details of the new coherence scheme, it is instructive to examine a simple model of the relationship between the performance of a full-map directory and the LimitLESS directory scheme. Let T_h be the average remote memory access latency for a full-map directory

Component	Name	Meaning
Memory	Read-Only	Some number of caches have read-only copies of the data.
	Read-Write	Exactly one cache has a read-write copy of the data.
	Read-Transaction	Holding read request, update is in progress.
	Write-Transaction	Holding write request, invalidation is in progress.
Cache	Invalid	Cache block may not be read or written.
	Read-Only	Cache block may be read, but not written.
	Read-Write	Cache block may be read or written.

Table 1: Directory states.

protocol. T_h includes factors such as the delay in the cache and memory controllers, invalidation latencies, and network latency. Given the hardware protocol latency T_h, it is possible to estimate the average remote memory access latency for the LimitLESS protocol with the formula: $T_h + mT_s$, where T_s (the software latency) is the average delay for the full-map directory emulation interrupt, and m is the fraction of memory accesses that overflow the small set of pointers implemented in hardware.

For example, our dynamic trace-driven simulations of a Weather Forecasting program running on 64 node Alewife memory system (see Section 5) indicate that $T_h \approx 35$ cycles. If $T_s = 100$ cycles, then remote accesses with the LimitLESS scheme will be 10% slower (on average) than with the full-map protocol when $m \approx 3\%$. Since the Weather program is, in fact, optimized such that 97% of accesses to remote data locations "hit" in the limited directory, the full-map emulation will cause a 10% delay in servicing requests for data.

LimitLESS directories are scalable, because the memory overhead grows as $O(N)$, and the performance approaches that of a full-map directory as system size increases. Although in a 64 processor machine, T_h and T_s are comparable, in much larger systems the internode communication latency will be much larger than the processors' interrupt handling latency ($T_h \gg T_s$). Furthermore, improving processor technology will make T_s even less significant. In such systems, the LimitLESS protocol will perform about as well as the full-map protocol, even if $m = 1$. This approximation indicates that if both processor speeds and multiprocessor sizes increase, handling cache coherence completely in software will become a viable option. In fact, the LimitLESS protocol is the first step on the migration path towards interrupt-driven cache coherence. Other systems [23] have also experimented with handling cache misses entirely in software.

3.2 Specification of the LimitLESS Scheme

In the above discussion, we assume that the hardware latency (T_h) is approximately equal for the full-map and the LimitLESS directories, because the LimitLESS protocol has the same state transition diagram as the full-map protocol. The memory controller side of this protocol is illustrated in Figure 2, which contains the memory states listed in Table 1. These states are mirrored by the state of the block in the caches, also listed in Table 1. It is the responsibility of the protocol to keep the states of the memory and the cache blocks coherent. The protocol enforces coherence by transmitting messages (listed in Table 3) between the cache/memory controllers. Every message contains the address of a memory block, to indicate which directory entry should be used when processing the message. Table 3 also indicate whether a message contains the data associated with a memory block.

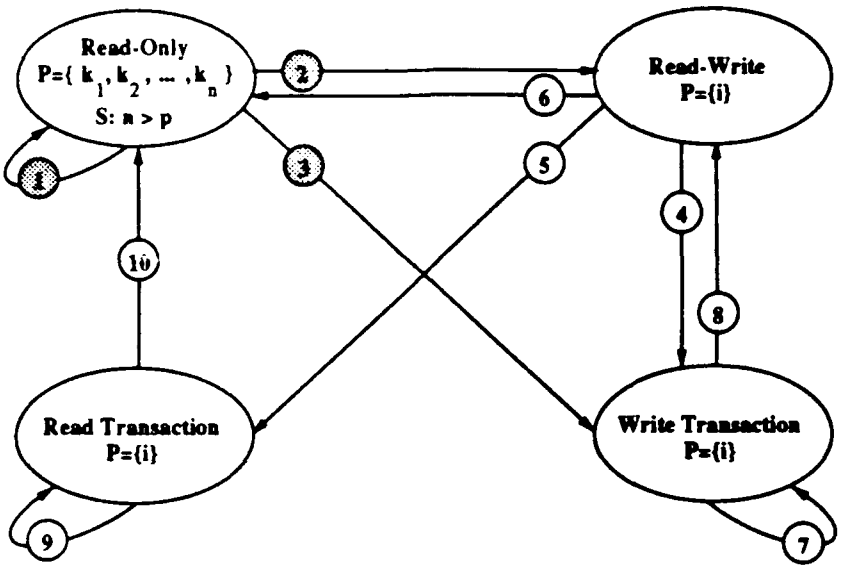

Figure 2: Directory state transition diagram for the full-map and LimitLESS coherence schemes.

The state transition diagram in Figure 2 specifies the states, the composition of the pointer set (P), and the transitions between the states. Each transition is labeled with a number that refers to its specification in Table 2. This table annotates the transitions with the following information: 1. The *input message* from a cache which initiates the transaction and the identifier of the cache that sends it. 2. A *precondition* (if any) for executing the transition. 3. Any *directory entry change* that the transition may require. 4. The *output message* or messages that are sent in response to the input message. Note that certain transitions require the use of an acknowledgment counter ($AckCtr$), which is used to ensure that cached copies are invalidated before allowing a write transaction to be completed.

For example, Transition 2 from the Read-Only state to the Read-Write state is taken when cache i requests write permission (WREQ) and the pointer set is empty or contains just cache i ($P = \{\}$ or $P = \{i\}$). In this case, the pointer set is modified to contain i (if necessary) and the memory controller issues a message containing the data of the block to be written (WDATA).[1].

Following the notation in [8], both full-map and LimitLESS are members of the $Dir_N NB$ class of cache coherence protocols. Therefore, from the point of view of the protocol specification, the LimitLESS scheme does not differ substantially from the full-map protocol. In fact, the LimitLESS protocol is also specified in Figure 2. The extra notation on the Read-Only ellipse ($S : n > p$) indicates that the state is handled in software when the size of the pointer set (n) is greater than the size of the limited directory (p). In this situation, the transitions with the shaded labels (1, 2, and 3) are executed by the interrupt handler on the processor that is local to the overflowing directory. When the protocol changes from a software-handled state to a hardware-handled state, the processor must modify the directory state so that the memory controller can resume responsibility for the protocol transitions.

[1]The Alewife machine will actually support an optimization of this transition that would sent a modify grant (MODG), rather than write data (WDATA). For the purposes of this paper, such optimizations have been eliminated in order to simplify the protocol specification.

Transition Label	Input Message	Precondition	Directory Entry Change	Output Message(s)
1	$i \rightarrow$ RREQ	—	$P = P \cup \{i\}$	RDATA $\rightarrow i$
2	$i \rightarrow$ WREQ	$P = \{i\}$	—	WDATA $\rightarrow i$
	$i \rightarrow$ WREQ	$P = \{\}$	$P = \{i\}$	WDATA $\rightarrow i$
3	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \notin P$	$P = \{i\}$, AckCtr $= n$	$\forall k$, INV $\rightarrow k$,
	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \in P$	$P = \{i\}$, AckCtr $= n - 1$	$\forall k, \neq i$ INV $\rightarrow k$,
4	$j \rightarrow$ WREQ	$P = \{i\}$	$P = \{j\}$	INV $\rightarrow i$
5	$j \rightarrow$ RREQ	$P = \{i\}$	$P = \{j\}$	INV $\rightarrow i$
6	$i \rightarrow$ REPM	$P = \{i\}$	$P = \{\}$	—
7	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ ACKC	AckCtr $\neq 1$	AckCtr = AckCtr $- 1$	—
	$j \rightarrow$ REPM	—	—	—
8	$j \rightarrow$ ACKC	AckCtr $= 1$, $P = \{i\}$	—	WDATA $\rightarrow i$
	$j \rightarrow$ UPDATE	$P = \{i\}$	—	WDATA $\rightarrow i$
9	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ REPM	—	—	—
10	$j \rightarrow$ UPDATE	$P = \{i\}$	—	RDATA $\rightarrow i$

Table 2: Annotation of the state transition diagram.

Type	Symbol	Name	Data?
Cache to Memory	RREQ	Read Request	
	WREQ	Write Request	
	REPM	Replace Modified	✓
	UPDATE	Update	✓
	ACKC	Invalidate Acknowledge	
Memory to Cache	RDATA	Read Data	✓
	WDATA	Write Data	✓
	INV	Invalidate	
	BUSY	Busy Signal	

Table 3: Protocol messages for hardware coherence.

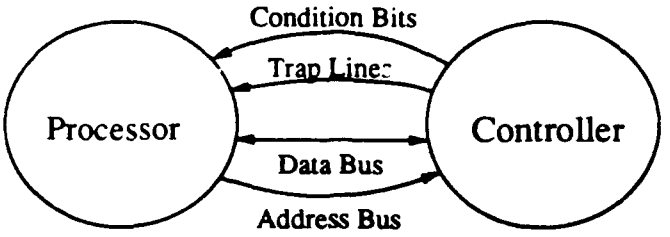

Figure 3: Signals Between Processor and Controller.

The Alewife machine will support an optimization of the LimitLESS protocol that maximizes the number of transactions that are serviced in hardware. When the controller interrupts the processor due to a pointer array overflow, the processor completely empties the pointer array into local memory. The fact that the directory entry is empty allows the controller to continue handling read requests until the next pointer array overflow. This optimization is called Trap-On-Write, because the memory controller must interrupt the processor upon a write request, even though it can handle read requests itself. The next section explains the mechanisms that are needed to implement the software/hardware hand-off required by the LimitLESS protocol.

4 Hardware Interfaces for LimitLESS

This section discusses the architectural properties and hardware interfaces needed to support the LimitLESS directory scheme. We describe how these interfaces are supported in the Alewife machine. Since the Alewife network interface is somewhat unique for shared-memory machines, it is examined in detail. Afterwards, we introduce the additional directory state that Alewife supports, over and above the state that is needed for a limited directory protocol, and examine its application to LimitLESS. Other uses for the extra states are discussed in Section 6.

To set the stage for this discussion, examine Figure 3. The hardware interface between the Alewife processor and controller consists of several elements. The address and data buses permit processor manipulation of controller state and initiation of actions via simple load and store instructions (memory-mapped I/O[2]). The controller returns two condition bits and several trap lines to the processor.

4.1 Necessary support for LimitLESS

To support the LimitLESS protocol efficiently, a cache-based multiprocessor needs several properties. First, it must be capable of rapid trap handling. Because LimitLESS is an extension of hardware through software, the LimitLESS protocol will not perform well on processors or software architectures that require hundreds of cycles to begin executing the body a trap handler. The Alewife machine employs a processor with register windows (SPARCLE) and a finely-tuned software trap architecture that permits trap code to begin execution within five to ten cycles from the time that a trap is initiated.

[2]The memory-mapped I/O space is distinguished from normal memory space by a distinct Alternate Space Indicator (ASI). In a way, the ASI bits are part of the address bus; see [20] for further details.

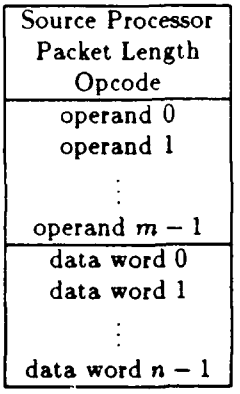

Figure 4: Uniform Packet Format for the Alewife Machine

Second, the processor needs complete access to coherence-related controller state such as pointers and state bits in the hardware directories. This state will be modified, when appropriate, by the LimitLESS trap handler. In Alewife, the directories are placed in a special region of memory that may be read and written by the processor.

Finally, a machine implementing the LimitLESS protocol needs an interface to the network that allows the processor to launch and intercept cache-coherence protocol packets. Most shared-memory multiprocessors export little or no network functionality to the processor; the Alewife machine is somewhat unique in this respect. Network access is provided through the Interprocessor-Interrupt (IPI) mechanism, which is discussed in the next section.

4.2 Interprocessor-Interrupt (IPI) in the Alewife machine

The Alewife machine supports a complete interface to the interconnection network. This interface provides the processor with a *superset* of the network functionality needed by the cache-coherence hardware. Not only can it be used to send and receive cache protocol packets, but it can also be used to send preemptive messages to remote processors (as in message-passing machines). The name Interprocessor-Interrupt (IPI) comes from the preemptive nature of messages that are directed to remote processors.

We stress that the IPI interface is a single generic mechanism for network access – *not* a conglomeration of different mechanisms. The power of such a mechanism lies in its generality.

Network Packet Structure To simplify the IPI interface, network packets have a single, uniform structure, shown in Figure 4. This figure includes only the information seen at the destination; routing information is stripped off by the network. The Packet Header contains the ID of the source processor, the length of the packet, and an opcode. It is a single word in the Alewife machine. Following the header are zero or more operands and data words. The distinction between operands and data is software-imposed; however, this is a useful abstraction supported by the IPI interface.

Opcodes are divided into two distinct classes: protocol and interrupt. Protocol opcodes are

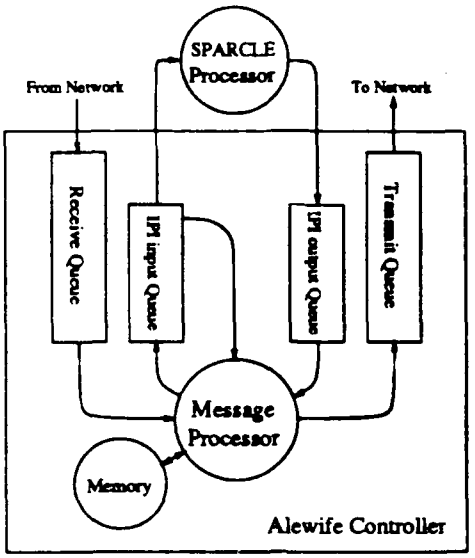

Figure 5: Simplified, Queue-based Diagram of the Alewife Controller

used for cache-coherence traffic; they are normally produced and consumed by the controller hardware, but also be produced or consumed by the LimitLESS trap-handler. Protocol opcodes encode the type of coherence transaction; for example, a read miss would generate a message with <opcode = RREQ>, <Packet Length = 2>, and <Operand0 = Address>. Packets with protocol opcodes are called protocol packets.

Interrupt opcodes have their MSBs set and are used for interprocessor messages. Their format is defined entirely by the software. Packets with interrupt opcodes are called interprocessor interrupts and are processed in software at their destinations.

Transmission of IPI packets A simplified, queue-based diagram of the internals of the Alewife controller is shown in Figure 5. This is a "memory-side" diagram; for simplicity it excludes the processor cache.

The processor interface uses memory-mapped store instructions to specify destination, opcode, and operands. It also specifies a starting address and length for the data portion of the packet. Taken together, this information completely specifies an outgoing packet. Note that operands and data are distinguished by their specification: operands are written explicitly through the interface, while data is fetched from memory. The processor initiates transmission by storing to a special trigger location, which enqueues the request on the *IPI output queue*.

Reception of IPI packets When the controller wishes to hand a packet to the processor, it places it in a special input buffer, the *IPI input queue*. This queue is large enough for several protocol packets and overflows into the network *receive queue*. The forwarding of packets to the IPI queue is accompanied by an interrupt to the processor.

The header (source, length, opcode) and operands of the packet at the head of the IPI input queue can be examined with simple load instructions. Once the trap routine has examined the

Meta State	Description
Normal	Directory being handled by hardware.
Trans-In-Progress	Interlock. Software processing in progress.
Trap-On-Write	Trap for WREQ, UPDATE, and REPM.
Trap-Always	Trap for all incoming packets

Table 4: Directory Meta States for the LimitLESS protocol

header and operands, it can either discard the packet or store it to memory, beginning at a specified location. In the latter case, the data that is stored starts from a specified offset in the packet. This store-back capability permits message-passing and block-transfers in addition to enabling the processing of protocol packets with data.

IPI input traps are synchronous, that is, they are capable of interrupting instruction execution. This is necessary, because the queue topology shown in Figure 5 is otherwise subject to deadlock. If the processor pipeline is being held for a remote cache-fill [3] and the IPI input queue overflows, then the receive queue will be blocked, preventing the load or store from completing. At this point, a synchronous trap must be taken to empty the input queue. Since trap code is stored in local memory, it may be executed without network transactions.

4.3 Meta States for the LimitLESS protocol

As noted in Section 3, the LimitLESS protocol consists of a series of extensions to the basic limited directory protocol. That section discussed circumstances under which the memory controller would invoke the software. Having discussed the IPI interface, we can examine the hardware support for LimitLESS in more detail.

This support consists of two components, *meta states* and *pointer overflow trapping*. Meta states are directory modes and are listed in Table 4. They may be described as follows:

- Coherence for memory blocks which are in Normal mode are handled by hardware. These are lines whose worker-sets are less than or equal to the number of hardware pointers.
- The Trans-In-Progress mode is entered automatically when a protocol packet is passed to software (by placing it in the IPI input queue). It instructs the controller to block on *all* future protocol packets for the associated memory block. The mode is cleared by the LimitLESS trap code after processing the packet.
- For memory blocks that are in the Trap-On-Write mode, read requests are handled as usual, but write requests (WREQ), update packets (UPDATE), and replace-modified packets (REPM) are forwarded to the IPI input queue. When packets are forwarded to the IPI queue, the directory mode is changed to Trans-In-Progress.
- Trap-Always instructs the controller to pass all protocol packets to the processor. As with Trap-On-Write, the mode is switched to Trans-In-Progress when a packet is forwarded to

[3] In the Alewife machine, we have the option of switching contexts on cache misses (see [19]). However, certain forward-progress concerns dictate that we occasionally hold the processor while waiting for a cache-fill.

the processor.

The two bits required to represent these states are stored in directory entries along with the states of Figure 2 and the five hardware pointers.

Controller behavior for pointer overflow is straightforward: when a memory line is in the Read-Only state and all hardware pointers are in use, then an incoming read request for this line (RREQ) will be diverted into the IPI input queue and the directory mode will be switched to Trans-In-Progress.

Local Memory Faults What about local processor accesses? A processor access to local memory that must be handled by software causes a *memory fault.* The controller places the faulting address and access type (i.e. read or write) in special controller registers, then invokes a synchronous trap.

A trap handler must alter the directory when processing a memory fault to avoid an identical fault when the trap returns. To permit the extensions discussed in Section 6, the Alewife machine reserves a one bit pointer in each hardware directory entry, called the *Local Bit.* This bit ensures that local read requests will never overflow a directory. In addition, the trap handler can set this bit after a memory fault to permit the faulting access to complete.

4.4 Use of Interfaces in LimitLESS Trap

A possible implementation of the LimitLESS trap handler is as follows: when an overflow trap occurs for the first time on a given memory line, the trap code allocates a full-map bit-vector in local memory. This vector is entered into a hash table. All hardware pointers are emptied and their corresponding bits are set in this vector. The directory mode is set to Trap-On-Write before the trap returns. When additional overflow traps occur, the trap code locates the full-map vector in the hash table, emptying the hardware pointers and setting their corresponding bits in this vector.

Software handling of a memory line terminates when the processor traps on an incoming write request (WREQ) or local write fault. The trap handler finds the full-map bit vector and empties the hardware pointers as above. Next, it records the identity of the requester in the directory, sets the acknowledgment counter to the number of bits in the vector that are set, and places the directory in the Normal mode, Write Transaction state. Finally, it sends invalidations to all caches with bits set in the vector. The vector may now be freed. At this point, the memory line has returned to hardware control. When all invalidation are acknowledged, the hardware will send the data with write permission to the requester.

Of course, this is only one of a number of possible LimitLESS trap handlers. Since the trap handler is part of the Alewife software system, many other implementations are possible.

5 Performance Measurements

This section describes some preliminary results from the Alewife system simulator that compare the performance of limited, LimitLESS, and full-map directories. The protocols are evaluated in terms of the total number of cycles needed to execute an application on a 64 processor Alewife

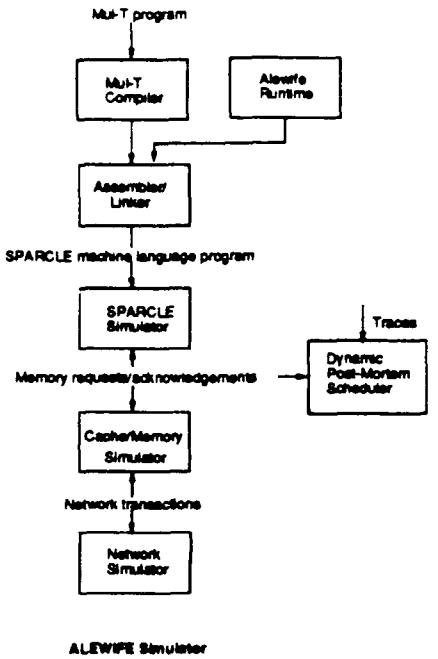

Figure 6: Diagram of ASIM, the Alewife system simulator.

machine. Using execution cycles as a metric emphasizes the bottom line of multiprocessor design: how fast a system can run a program.

5.1 The Measurement Technique

The results presented below are derived from complete Alewife machine simulations and from dynamic post-mortem scheduler simulations. Figure 6 illustrates these two branches of ASIM, the Alewife Simulator.

ASIM models each component of the Alewife machine, from the multiprocessor software to the switches in the interconnection network. The complete-machine simulator runs programs that are written in the Mul-T language [24], optimized by the Mul-T compiler, and linked with a runtime system that implements both static work distribution and dynamic task partitioning and scheduling. The code generated by this process runs on ASIM, the Alewife machine simulator, which consists of processor, cache/memory, and network modules.

Although the memory accesses in ASIM are usually derived from applications running on the SPARCLE processor, ASIM can alternatively derive its input from a dynamic post-mortem trace scheduler, shown on the right side of Figure 6. Post-mortem scheduling is a technique that generates a parallel trace from a uniprocessor execution trace that has embedded synchronization information [25]. The post-mortem scheduler is coupled with the memory system simulator and incorporates feedback from the network in issuing trace requests, as described in [26]. The use of this input source is important because it lets us expand the workload set to include large parallel applications written in a variety of styles.

As shown in Figure 6, both the full-machine and the dynamic post-mortem simulations use

the same cache/memory and network simulation modules. The cache/memory simulator can be configured to run a number of different coherence schemes, including software-enforced coherence and a scheme that only caches private data. In addition, the memory simulator can vary more basic parameters such as cache size and block size. The network simulator can model both circuit and packet switching interconnects, with either mesh or Omega topologies.

The simulation overhead for large machines forces a trade-off between application size and simulated system size. Programs with enough parallelism to execute well on a large machine take an inordinate time to simulate. When ASIM is configured with its full statistics-gathering capability, it runs at about 5000 processor cycles per second on an unloaded SPARCserver 330. At this rate, a 64 processor machine runs approximately 80 cycles per second. Most of the simulations that we chose for this paper run for one million cycles (a fraction of a second on a real machine), which takes 3.5 hours to complete. This lack of simulation speed is one of the primary reasons for implementing the Alewife machine in hardware — to enable a thorough evaluation of our ideas.

For the purpose of evaluating the potential benefits of the LimitLESS coherence scheme, we implemented an approximation of the new protocol in ASIM. The technique assumes that the overhead of the LimitLESS full-map emulation interrupt is approximately the same for all memory requests that overflow a directory entry's pointer array. This is the T_s parameter described in Section 3. During the simulations, ASIM simulates an ordinary full-map protocol. But when the simulator encounters a pointer array overflow, it stalls the both the memory controller and the processor that would handle the LimitLESS interrupt for T_s cycles. While this evaluation technique only approximates the actual behavior of the fully-operational LimitLESS scheme, it is a reasonable method for determining whether to expend the greater effort needed to implement the complete protocol.

5.2 Performance Results

Figure 7 presents the performance of a statically scheduled multigrid relaxation program on a 64-processor Alewife machine. This program was written in Mul-T and runs on a complete-machine simulation. The vertical axis on the graph displays several coherence schemes, and the horizontal axis shows the program's total execution time (in millions of cycles). All of the protocols, including the four-pointer limited directory (Dir_4NB), the full-map directory, and the LimitLESS scheme with full-map emulation latencies of 50 and 100 cycles (T_s = 50 and T_s = 100) require approximately the same time to complete the computation phase. This confirms the assumption that for applications with small worker-sets, such as multigrid, the limited (and therefore the LimitLESS) directory protocols perform almost as well as the full-map protocol. See [10] for more evidence of the general success of limited directory protocols.

A weather forecasting program, simulated with the dynamic post-mortem scheduling method, provides a case-study of an application that has not been completely optimized for limited directory protocols. Although the simulated application uses software combining trees to distribute its barrier synchronization variables, Weather has one variable that is initialized by one processor and then read by all of the other processors. Our simulations show that if this variable is flagged as read-only data, then a limited directory performs just as well for Weather as a full-map directory.

However, it is easy for a programmer to forget to perform such optimizations, and there

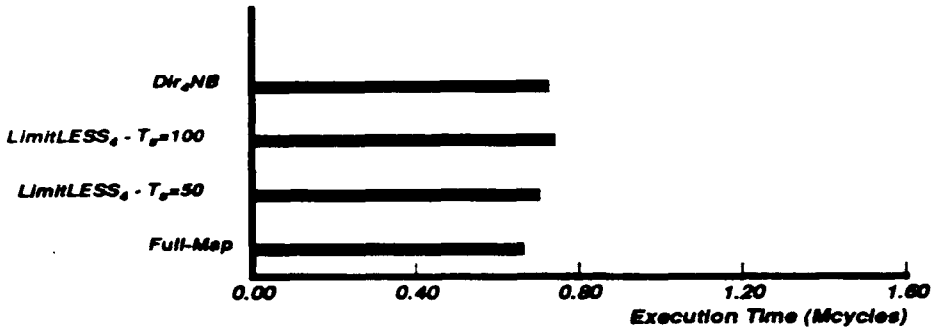

Figure 7: Static Multigrid, 64 Processors

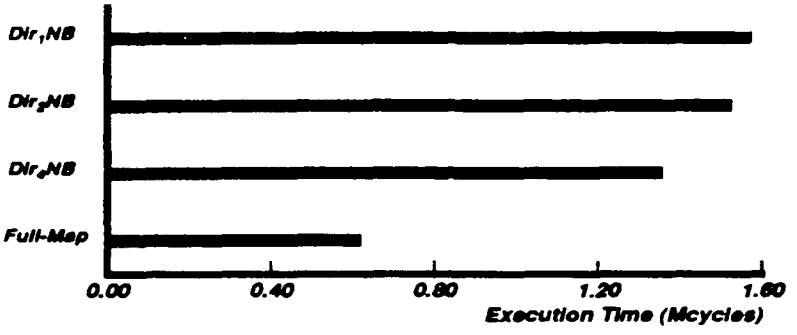

Figure 8: Weather, 64 Processors, limited and full-map directories.

are some situations where it is very difficult to avoid this type of sharing. Figure 8 gives the execution times for Weather when this variable is not optimized. The results show that when the worker-set of a single location in memory is much larger than the size of a limited directory, the whole system may suffer from hot-spot access to this location. So, limited directory protocols are extremely sensitive to the size of a heavily-shared data block's worker-set. If a multiprocessor's software is not perfectly optimized, limited directory thrashing may negate the benefits of caching shared data.

The effect of the unoptimized variable in Weather was not evident in previous evaluations of directory-based cache coherence [10], because the network model did not account for hot-spot behavior. Since the program can be optimized to eliminate the hot-spot, the new results do not contradict the conclusion of [10] that system-level enhancements make large-scale cache-coherent multiprocessors viable. Nevertheless, the experience with the Weather application reinforces the belief that complete-machine simulations are necessary to evaluate the implementation of cache coherence.

As shown in Figure 9, the LimitLESS protocol avoids the sensitivity displayed by limited directories. This figure compares the performance of a full-map directory, a four-pointer limited directory (Dir_4NB), and the four-pointer LimitLESS (LimitLESS$_4$) protocol with several values for the additional latency required by the LimitLESS protocol's software (T_s = 25, 50, 100, and

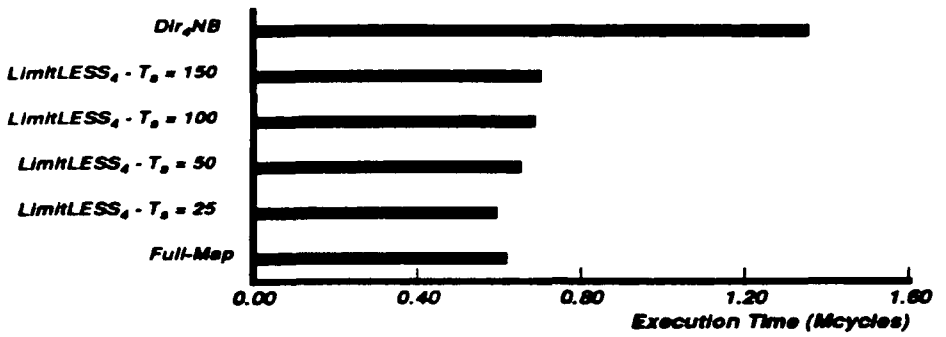

Figure 9: Weather, 64 Processors, LimitLESS with 25 to 150 cycle directory emulation latencies.

150). The execution times show that the LimitLESS protocol performs about as well as the full-map directory protocol, even in a situation where a limited directory protocol does not perform well. Furthermore, while the LimitLESS protocol's software should be as efficient as possible, the performance of the LimitLESS protocol is not strongly dependent on the latency of the full-map directory emulation. The current estimate of this latency in the Alewife machine is between 50 and 100 cycles.

It is interesting to note that the LimitLESS protocol, with a 25 cycle emulation latency, actually performs better than the full-map directory. This anomalous result is caused by the participation of the processor in the coherence scheme. By interrupting the Weather application software and slowing down certain processors, the LimitLESS protocol produces a slight back-off effect that reduces contention in the interconnection network.

The number of pointers that a LimitLESS protocol implements in hardware interacts with the worker-set size of data structures. Figure 10 compares the performance of Weather with a full-map directory, a limited directory, and LimitLESS directories with 50 cycle emulation latency and one ($LimitLESS_1$), two ($LimitLESS_2$), and four ($LimitLESS_4$) hardware pointers. The performance of the LimitLESS protocol degrades gracefully as the number of hardware pointers is reduced. The one-pointer LimitLESS protocol is especially bad, because some of Weather's variables have a worker-set that consists of exactly two processors.

This behavior indicates that multiprocessor software running on a system with a LimitLESS protocol will require some of the optimizations that would be needed on a system with a limited directory protocol. However, the LimitLESS protocol is much less sensitive to programs that are not perfectly optimized. Moreover, the software optimizations used with a LimitLESS protocol should not be viewed as extra overhead caused by the protocol itself. Rather, these optimizations might be employed, regardless of the cache coherence mechanism, since they tend to reduce hot-spot contention and to increase communication locality.

6 Extensions to the LimitLESS Scheme

Using the interface described in Section 4, the LimitLESS protocol may be extended in several ways. The simplest type of extension uses the LimitLESS trap handler to gather statistics about

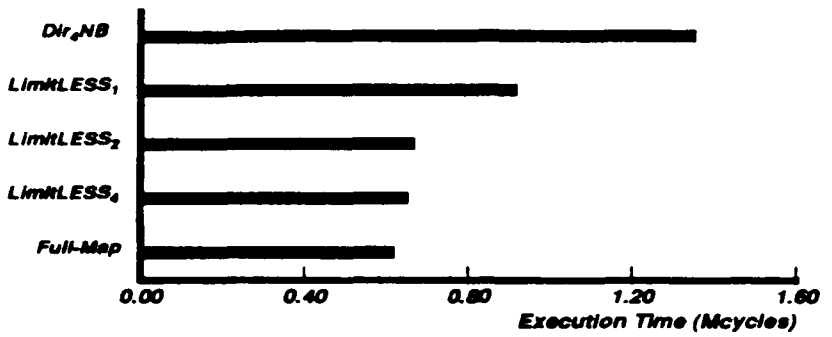

Figure 10: Weather, 64 Processors, LimitLESS scheme with 1, 2, and 4 hardware pointers.

shared memory locations. For example, the handler can record the worker-set of each variable that overflows its hardware directory. This information can be fed back to the programmer or compiler to help recognize and minimize the use of such variables. For studies of data sharing, a number of locations can be placed in the Trap-Always directory mode, so that they are handled entirely in software. This scheme permits complete profiling of memory transactions to these locations without degrading performance of non-profiled locations.

More interesting enhancements couple the LimitLESS protocol with the compiler and run-time systems to implement various special synchronization and coherence mechanisms. Previous studies such as [27] have examined the types of coherence which are appropriate for varying data types. The Trap-Always and Trap-On-Write directory modes (defined in Section 4) can be used to synthesize some of these coherence types. For example, the LimitLESS trap handler can cause FIFO directory eviction for data structures that are known to migrate from processor to processor. A FIFO lock data type provides another example; the trap handler can buffer write requests for a programmer-specified variable and grant the requests on a first-come, first-serve basis. The directory trap modes can also be used to construct objects that update (rather than invalidate) cached copies after they are modified.

The mechanisms that we propose to implement the LimitLESS directory protocol provide the type of generic interface that can be used for many different memory models. Judging by the number of synchronization and coherence mechanisms that have been defined by multiprocessor architects and programmers, it seems that there is no lack of uses for such a flexible coherence scheme.

7 Conclusion

This paper proposed a new scheme for cache coherence, called LimitLESS, which is being implemented in the Alewife machine. Hardware requirements include rapid trap handling and a flexible processor interface to the network. Preliminary simulation results indicate that the LimitLESS scheme approaches the performance of a full-mapped directory protocol with the memory efficiency of limited directory protocol. Furthermore, the LimitLESS scheme provides a migration path toward a future in which cache coherence is handled entirely in software.

8 Acknowledgments

In one way or another, all of the members of the Alewife group at MIT helped develop and evaluate the ideas presented in this paper. In particular, David Kranz wrote the Mul-T compiler, Beng-Hong Lim and Dan Nussbaum wrote the SPARCLE simulator and run-time system, Kirk Johnson supported the benchmarks, Kiyoshi Kurihara found the hot-spot variable in the weather forecasting code, and Gino Maa wrote the network simulator.

The notation for the transition state diagram borrows from the doctoral thesis of James Archibald at the University of Washington, Seattle, and from work done by Ingmar Vuong-Adlerberg at MIT.

Pat Teller of NYU provided the source code of the Weather application. Harold Stone and Kimming So helped us obtain the Weather trace. The post-mortem scheduler was implemented by Mathews Cherian with Kimming So at IBM. It was extended by Kiyoshi Kurihara to include several other forms of barrier synchronization such as backoffs and software combining trees, and to incorporate feedback from the network.

Machines used for simulations were donated by SUN Microsystems and Digital Equipment Corporation. The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, and by grants from the Sloan Foundation and IBM.

References

[1] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, IEEE, New York, June 1983.

[2] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[3] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 348–354, IEEE, New York, June 1985.

[4] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, IEEE, New York, June 1985.

[5] James Archibald and Jean-Loup Baer. An Economical Solution to the Cache Coherence Problem. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 355–362, IEEE, New York, June 1985.

[6] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[7] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[8] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[9] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, 74–77, June 1990.

[10] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[11] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[12] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[13] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel tasks. In *Proceedings of Symposium on Lisp and Functional Programming*, June 1990. To appear.

[14] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[15] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 9–21, February 1988.

[16] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[18] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[19] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[20] SPARC Architecture Manual. 1988. SUN Microsystems, Mountain View, California.

[21] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12), December 1984.

[22] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, 1987.

[23] David R. Cheriton, Gert A. Slavenberg, and Patrick D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 367–374, IEEE, New York, June 1986.

[24] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implemenation*, June 1989.

[25] Mathews Cherian. *A Study of Backoff Barrier Synchronization in Shared-Memory Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[26] Kiyoshi Kurihara. *Performance Evaluation of Large-Scale Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1990.

[27] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

Adaptive Routing using Virtual Channels [1]

William J. Dally and Hiromichi Aoki
Artificial Intelligence Laboratory
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

July 3, 1990, revised September 16, 1990

Abstract

The use of adaptive routing in a multicomputer interconnection network improves network performance by making use of all available paths and provides fault tolerance by allowing messages to route around failed channels and nodes. This paper describes two deadlock-free adaptive routing algorithms. Both algorithms allocate virtual channels using a count of the number of *dimension reversals* a packet has performed to eliminate cycles in resource dependency graphs. The *static algorithm* eliminates cycles in the network channel dependency graph. The *dynamic algorithm* improves virtual channel utilization by permitting dependency cycles and instead eliminating cycles in the packet wait-for graph. We prove that these algorithms are deadlock-free and give experimental measurements of their performance. For non-uniform traffic patterns, these algorithms improve network throughput by a factor of three compared to deterministic routing. The dynamic algorithm gives better performance at moderate traffic rates but requires source throttling to remain stable at high traffic rates. Both algorithms allow the network to gracefully degrade in the presence of faulty channels.

Keywords: Interconnection Networks, communication networks, packet routing, flow control, concurrent computing, parallel processing, multicomputers.

1 Introduction

1.1 Interconnection Networks

Interconnection networks are used to pass messages containing data and synchronization information between the nodes of concurrent computers [2, 16, 20, 7]. The messages may be sent between the processing nodes of a message-passing multicomputer [2] or

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825 and in part by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation.

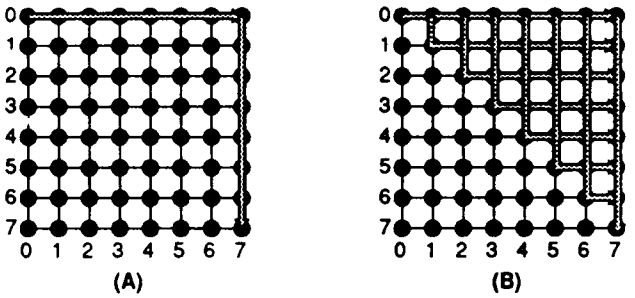

Figure 1: Routing packets in an 8-ary 2-cube from (i,0) to (7,i). (A) Using dimension order routing, seven packets must traverse the channel from (6,0) to (7,0). (B) Using adaptive routing, all packets proceed simultaneously increasing throughput by a factor of 7.

between the processors and memories of a shared-memory multiprocessor [16]. The interconnection network is often the critical component of a large parallel computer because performance is very sensitive to network latency and throughput and because the network accounts for a large fraction of the cost and power dissipation of the machine.

An interconnection network is characterized by its topology, routing, and flow control [9]. The topology of a network is the arrangement of its nodes and channels into a graph. Routing determines the path chosen by a message in this graph. Flow control deals with the allocation of channel and buffer resources to a message as it travels along this path. This paper deals with routing and flow control. Specifically, it is concerned with adaptive routing, a method for choosing a path through a network depending on the current state of the network. The methods described here are applicable to any topology; however, the examples in this paper consider their application to k-ary n-cube interconnection networks [10].

1.2 The Problem

Most existing routing networks [16, 20, 7] use deterministic routing. With deterministic routing, the path followed by a packet is determined solely by the source and destination of the message. If any channel along this path is heavily loaded, the packet will be delayed. If any channel along this path is faulty the packet cannot be delivered. A common deterministic routing algorithm is dimension-order routing where the packet routes in one dimension at a time, arriving at the proper coordinate in each dimension before proceeding to the next dimension.

Adaptive routing improves both the performance and fault tolerance of an interconnection

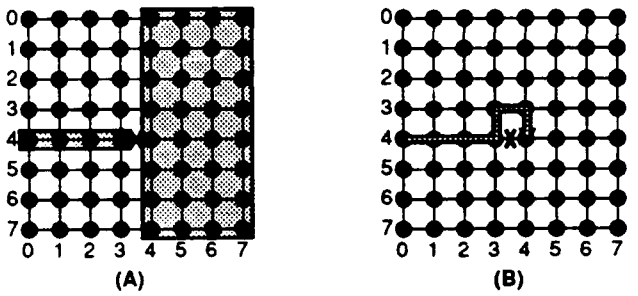

Figure 2: An 8-ary 2-cube network with a faulty channel from (3,4) to (4,4). (A) With dimension order routing, packets from the shaded area on the left to the shaded area on the right cannot be delivered. (B) Using adaptive routing packets can be delivered between all pairs of nodes.

network. Figure 1 shows a 8-ary 2-cube in which the node with coordinate (i,0) sends a packet to the node with coordinate (7,i). With dimension order deterministic routing (Figure 1A), seven of the eight packets must traverse the channel from (6,0) to (7,0). Thus only one of these seven packets can proceed at a time. With adaptive routing (Figure 1B) all of the packets can proceed simultaneously using alternate paths. For the traffic pattern shown in this example, adaptive routing increases throughput by a factor of seven.

Figure 2 shows the same network with a faulty channel from (3,4) to (4,4). With dimension-order deterministic routing, packets from node (i,4) to node (j,k) where $i \leq 3$ and $j \geq 4$ cannot be delivered. With adaptive routing, all messages can be delivered by routing around the faulty channel.

1.3 Adaptive Routing with Virtual Channels

Adaptive routing must be performed in a manner that is deadlock-free. Deadlock in an interconnection network occurs whenever there is a cyclic dependency for resources. For example, Figure 3 shows two messages deadlocked because each needs access to a channel currently occupied by the other.

Networks that use dimension order routing avoid deadlock by ordering channels so that messages travel along paths of strictly increasing channel numbers [13]. Channels are ordered so that all of the channels in each dimension are greater than all of the channels in the preceding dimension. This ordering eliminates cycles in the channel dependency graph and thus prevents deadlock. The ordering also prevents the use of adaptive routing since restricting dimension changes to be monotonic eliminates alternate paths.

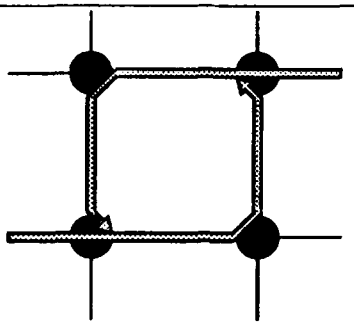

Figure 3: Two messages may become deadlocked if each is waiting on a resource (in this case a channel) held by the other.

This paper presents two deadlock-free adaptive routing algorithms. Both algorithms permit misrouting and avoid deadlock using virtual channels to eliminate cyclic dependencies. The *static algorithm* eliminates cycles in the channel dependency graph by numbering virtual channels and routing packets to traverse virtual channels in increasing order. It permits *dimension reversals*, routing from a higher dimension to a lower dimension, by using virtual channels. Each packet is labeled with a dimension reversal number that is initialized to zero. Each time a packet performs a dimension reversal, its dimension reversal number is incremented and it is routed on a *class* of virtual channels used only by packets with the same dimension reversal number. The number of classes of virtual channels places an upper limit on the maximum number of dimension reversals permitted. Once a packet has made this number of dimension reversals, it is restricted to dimension order routing.

The *static algorithm* restricts the number of dimension reversals permitted and makes inefficient use of the pool of virtual channels. A packet may be blocked waiting for a virtual channel in its dimension reversal class, while many other virtual channels for the same physical channel remain idle.

The *dynamic algorithm* overcomes these limitations by permitting cycles in the channel dependency graph. Deadlock is avoided by eliminating cycles from the packet *wait-for graph*. As with the static algorithm, packets are labeled with their dimension reversal number. Packets may use any class virtual channel during routing; however they are not permitted to wait for a virtual channel held by a packet with a lower dimension reversal number. If all available virtual channels are occupied by packets with lower dimension reversal numbers, the packet reverts to dimension order routing on a set of virtual channels reserved for this purpose.

The dynamic algorithm places no restrictions on the number of dimension reversals

permitted[2]. This algorithm also makes more efficient use of virtual channels by allowing any packet to use any idle virtual channel.

1.4 Related Work

Virtual channels were introduced in [13] for purposes of deadlock avoidance. This paper showed how a cyclic network can be made deadlock-free by restricting routing so there are no cycles in the channel dependency graph and then adding virtual channels to reconnect the network. Virtual channels have also been used to support multiple virtual circuits [3], and to increase network throughput [11].

A deadlock-free adaptive routing algorithm based on virtual channels is described in [8], [9], and [17]. However, this algorithm does not permit misrouting (routing a packet along a non-shortest path) and thus cannot route around certain network faults. An adaptive wormhole routing algorithm that permits misrouting is described in [18]; however, this algorithm is not deadlock-free. Ngai and Seitz [19] describe an adaptive routing algorithm for store-and-forward networks based on packet exchange. Another store-and-forward adaptive routing algorithm based on promotion is described in [1]. These two algorithms require that entire packets be buffered and thus cannot be used with wormhole routing. Wide area networks often use table-based adaptive routing algorithms [21]. Circuit-switched networks have used adaptive routing algorithms based on tree search [5].

1.5 Outline

The next section introduces the notation, terminology, and assumptions that will be used throughout this paper. Section 3 describes the two deadlock-free adaptive routing algorithms in more detail and proves that they are deadlock free. The results of experiments using these algorithms are described in Section 4.

2 Preliminaries

2.1 Topology

An interconnection network is a strongly-connected, directed graph, $I = G(N, C)$. The vertices of I are a set of *nodes*, N. The edges are a set of *channels*, $C \subseteq N \times N$. Each

[2]As a practical matter, the number of dimension reversals will be limited by the size of the packet header field used to hold the packet's dimension reversal number.

channel is unidirectional and carries data from a source node to a destination node. A bidirectional network is one where $(n1, n2) \in C \Rightarrow (n2, n1) \in C$.

2.2 Flow Control

Communication between nodes is performed by sending *messages*. A message may be broken into one or more *packets* for transmission. A packet is the smallest unit of information that contains routing and sequencing information. A packet contains one or more flow control digits or *flits*. A flit is the smallest unit on which flow control is performed. Information is transferred over physical channels in physical transfer units or *phits*. A phit is usually the same size or smaller than a flit.

Each physical channel, $c_i \in C$, in the network is composed of one or more virtual channels, $c_{ij} \in C'$. The virtual channels associated with a single physical channel share physical channel bandwidth, allocated on a flit by flit basis. However, each virtual channel contains its own queue and is allocated on a packet by packet basis independently of the other virtual channels. For purposes of deadlock analysis, each virtual channel is logically a separate channel.

2.3 Routing

A packet is assigned a route through the network according to a *routing relation*, $R \subseteq C' \times N \times C'$. Given the virtual channel occupied by the head of the packet and the destination node of the packet, the routing relation specifies a (possibly singleton) set of virtual channels on which the packet may be routed.

A selection function, $\rho(P(C'), \alpha) \mapsto C'$, is used to pick the next channel of the route from the elements of this set using some additional information, α. This additional information may include the occupancy and/or operational status of channels in the network. The next channel selected for a packet, p_i, is denoted next(p_i).

The *channel dependency graph* for an interconnection network, I, and routing relation, R is a directed graph, $D = G(C', E)$. Its vertices are C', the virtual channels of I, and its edges are given by the projection of the routing relation onto $C' \times C'$:

$$E = \{(c_i, c_j) | (c_i, n, c_j) \in R \text{ for some } n \in N\}. \quad (1)$$

Consider a network, I, occupied by a set of packets, P, where each packet, $p_i \in P$, occupies a particular set of virtual channels, $\text{occ}(p_i) \subseteq C'$. The *wait for graph* of I is a directed graph, $W = G(P, E_W)$. The vertices of W are P, the set of packets in the

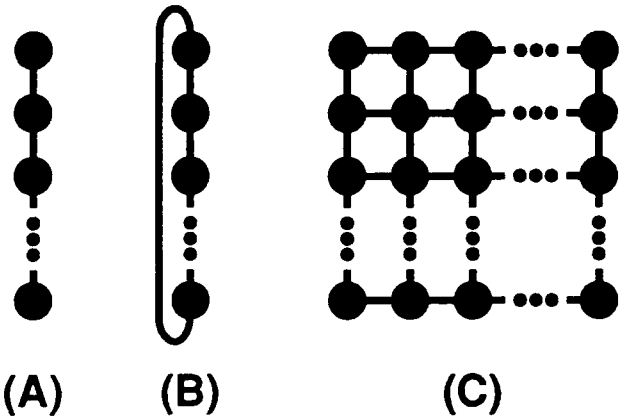

Figure 4: A k-ary n-cube is a cube of n dimensions with k nodes in each dimension. (A) Meshes are cubes with each dimension connected in a linear array. (B) Connecting each dimension in a ring gives a Tori. (C) Higher dimensional cubes are constructed by combining like elements of lower dimensional cubes.

network at a given instance of time. There is an edge of W, $e_i \in E_W$, for each packet that is waiting on another packet to acquire a resource:

$$E_W \cdot \{(p_i, p_j) | \text{next}(p_i) \in \text{occ}(p_j)\}. \tag{2}$$

2.4 Performance

The performance of a fault-free network is measured in terms of its *latency*, T, and its *throughput*, λ_{sat}. The latency of a message is the elapsed time from when the message send is initiated until the message is completely received. Network latency is the average message latency under specified conditions. Network throughput is the number of messages the network can deliver per unit time.

The performance of a faulty network is measured in terms of latency, throughput, and node loss. The latency and throughput of a network degrade as channels fail. The rate at which they degrade gives a figure of merit for the network. Node loss is the fraction of fault-free nodes that become disconnected from some other fault free node.

2.5 k-ary n-cube Networks

A k-ary n-cube is a radix k cube with n dimensions having $N = k^n$ nodes. The radix

implies that there are k-nodes in each dimension. The nodes in each dimension may be connected in a linear array giving a mesh network (Figure 4A) or in a ring giving a torus network (Figure 4B). A k-ary 1-cube (Figure 4A,B) is a k node ring or linear array. A k-ary 2-cube (Figure 4C) is constructed by taking k k-ary 1-cubes and connecting like elements. In general, a k-ary n-cube is constructed from k k-ary $(n-1)$-cubes by connecting like elements into rings or linear arrays.

Every node has an address that is an n digit, radix k number, $a_{n-1} \ldots a_0$. Each address digit, a_d, represents a node's coordinate in dimension d and can take on values in the range $[0, k-1]$. In a torus network, nodes are connected to all nodes with an address that differs in only one digit by ± 1 mod k. In a mesh, nodes are connected to all nodes with an address that differs in one digit by ± 1 where the result is in the range $[0, k-1]$.

The dimensions and directions of the cube partition the set of virtual channels, C', into subsets for each dimension: $C'_{00}, C'_{01} \ldots, C'_{(n-1)0}, C'_{(n-1)1}$. A channel, c_i, with source, n_s, and destination, n_d, whose addresses differ in the d^{th} position is said to be a channel in the d^{th} dimension. If $n_s > n_d$, $c_i \in C'_{d0}$. If $n_s < n_d$, $c_i \in C'_{d1}$. This definition partitions the two directions of a given direction into distinct channel sets.

Many networks are included in the family of k-ary n-cubes. At the extreme of $k = 2$, we have a binary n-cube. At the extreme of $n = 1$ we have a ring or linear array. For $n = 2$ we have a torus or 2D mesh. These networks have been used in several message passing computers [20, 6, 7]. For the remainder of this paper we consider only mesh-connected k-ary n-cubes.

3 Adaptive Routing Algorithms

This section describes two deadlock-free adaptive routing algorithms that use a packet's dimension reversal number to avoid cycles in resource dependency graphs. First dimension reversals are defined and the static and dynamic algorithms for assigning virtual channels to packets are presented and proved deadlock free. Finally, strategies for selecting among admissible channels that guarantee progress are described.

3.1 Dimension Reversals

The dimension reversal number of a packet is the count of the number of times a packet has routed from a channel in one dimension, p, to a channel in a lower dimension, $q < p$. Dimension reversal (DR) numbers are assigned to packets as follows:

1. All packets are initialized with a DR of 0.

2. Each time a packet routes from a channel $c_i \in C'_p$ to a channel $c_j \in C'_q$ where $p > q$ the DR of a packet is incremented.

3.2 The Static Algorithm

The static algorithm divides the virtual channels of each physical channel into non-empty classes numbered zero to r, where r is the maximum number of dimension reversals permitted. Packets with DR $< r$ may route in any direction but must use only virtual channels of class DR. Once a packet has DR$= r$, it must use dimension order routing on the virtual channels of class r. Thus, when a packet makes its final dimension reversal, it must start routing in the lowest dimension in which its current node address differs from the destination address.

Assertion 1: The Static Algorithm is deadlock free.

Proof: The channel dependency graph is acyclic. Assign a number, num(c_i), to each channel, c_i, in each dimension, C_{dr}, so channel numbers increase in the direction of routing[3]. Now order all virtual channels according to their class, dimension, and number. With this ordering, packets using the static algorithm will always traverse channels in ascending order. Thus the channel dependency graph is acyclic and the routing is deadlock free. ∎

3.3 The Dynamic Algorithm

The dynamic algorithm divides the virtual channels of each physical channel into two non-empty classes: adaptive and deterministic. Packets originate in the adaptive channels. While in these channels, they may route in any direction and there is no maximum limit on the number of dimension reversals a packet may make. However, a packet with a DR of p cannot wait on a channel currently occupied by a packet with a DR of q if $p \geq q$. A packet that reaches a node where all permissible output channels are occupied by packets with equal or lower DRs must switch to the deterministic class of virtual channels[4]. When a packet enters the deterministic channels, it must start routing in the lowest dimension in which the current node address and the destination address differ. Once on the deterministic channels, the packet must route in dimension order and cannot reenter the adaptive channels.

Assertion 2: The dynamic algorithm is deadlock free.

Proof: By contradiction. If the network is deadlocked, then there is a set of packets, P, waiting on resources (virtual channels) held by other packets in P. There exists a packet,

[3]This method can be extended to tori by using the method given in [13].

[4]A packet may wait for a finite amount of time before resorting to dimension order routing.

p_{max}, such that the $DR(p_{max}) \geq DR(q) \; \forall \; q \in P$. However, p_{max} cannot be blocked since it is not allowed to wait on a resource held by any packet with lower or equal DR. ∎

The dynamic algorithm is deadlock free even though it permits cycles in the network's channel dependency graph. This does not contradict Theorem 1 of [13] as that theorem assumes deterministic routing. In [13], R is a function, not a relation, so if a cycle exists in the channel dependency graph, a packet is required to follow the cycle. With adaptive routing, R is a relation. There may be many channels available to route a packet. Deadlock can be avoided by choosing a channel that does not create a cycle in the packet wait-for graph.

3.4 Routing Policy

The static and dynamic algorithms specify a routing relation, R, that is guaranteed to be deadlock-free. For the static algorithm, R is statically determined from network topology. For the dynamic case, R is determined by the current channel occupancy as well as topology. Both algorithms leave open the choice of a selection function, ρ, to choose the next channel of a route from among the permissible channels defined by R. A selection function must be concerned with issues of progress and throttling.

Progress

The static and dynamic algorithms allow a packet to route deadlock-free along an arbitrary path in a k-ary n-cube network. These algorithms by themselves, however, give no guarantee that a packet will ever reach its destination.

To guarantee progress toward a destination, misrouting must be limited. Simply prohibiting misrouting is too restrictive because it prevents single-dimension messages from routing around faults and congestion. A simple method for limiting misrouting is to place an upper limit on the number of steps a message may take away from its destination. Then it is easy to prove progress by showing that a weighted sum of the distance to the destination and the number of misrouting steps remaining is strictly decreasing. A variant on this scheme is to limit the ratio of misrouting steps to progress steps – e.g., no more than one step back for each two steps forward.

Throttling

The adaptive virtual channel pool of the dynamic algorithm can be monopolized by eager sources unless some form of throttling [4] is used. If many sources attempt to inject messages into the network faster than the network is able to handle them, these

new messages will consume all available virtual channels in the adaptive pool. Older messages will be forced to revert to deterministic routing using the deterministic pool.

Throttling can be performed by using a hybrid of the static and dynamic algorithms. The virtual channels are divided into classes as in the static algorithm. A packet with a DR of p is permitted to select a channel of class q only if (1) $p \geq q$ and (2) there is no packet with a DR greater than p in a virtual channel of the same physical channel of class q or less. This method divides the adaptive virtual channel pool into classes to prevent new methods from consuming the entire pool. In practice, two classes, 0 and 1, are sufficient to limit the channels consumed by injected messages.

Selection Functions

There are many possible selection functions. A few possibilities are shown below. Each selection function is shown as a list of directives. For each arriving packet, the router attempts the directives in order. If there is no channel available for a directive to be applied the router moves on to the next directive in the list after a (possibly null) timeout period. The performance of different selection functions is evaluated by experiment below.

- Favor minimum congestion:
    1. Pick the dimension with the most available virtual channels that moves the packet toward its destination.
    2. Misroute if permitted.
    3. Revert to deterministic routing.
- Favor routing flexibility:
    1. Pick the dimension with the greatest distance to travel and route in the proper direction of that dimension.
    2. Route in the proper direction of any dimension.
    3. Misroute if permitted.
    4. Revert to deterministic routing.
- Favor straight lines:
    1. If not at the proper coordinate in the current dimension continue routing in the current dimension.
    2. Route in the proper direction of any dimension.
    3. Misroute if permitted.
    4. Revert to deterministic routing.

4 Experimental Results

To measure the performance of the adaptive routing algorithms described above, we have simulated a number of k-ary n-cube networks varying the routing relation, selection function, and traffic patterns. Faulty networks were simulated to measure performance degradation.

The simulator is a 9000-line C program that simulates interconnection networks at the flit-level. A flit transfer between two nodes is assumed to take place in one time unit. The network is simulated synchronously, moving all flits that have been granted channels in one time step and then advancing time to the next step. The simulator is programmable as to topology, routing algorithm, and traffic pattern.

All of the results in this section are for 256-node 16-ary 2-cube mesh networks with 16 virtual channels per physical channel.

4.1 Latency

Latency is measured by applying a constant rate source to each input and measuring the time from packet creation until the last flit of the packet is accepted at the destination. Source queueing time is included in the latency measurement.

Figure 5 compares the performance of deterministic dimension-order routing with static and dynamic adaptive routing under uniform random traffic. The figure shows latency as a function of offered traffic for the three routing strategies. Both adaptive routing strategies used a selection function that favors minimum congestion, and both permit misrouting. For deterministic routing, saturation occurs at 94% capacity. For static and dynamic routing, saturation occurs at 78% and 88% respectively.

Random traffic loads the network channels and buffers uniformly. Thus, adaptive routing affects performance only slightly for this traffic pattern. For small loads, adaptive routing slightly reduces latency by moving packets that would otherwise be blocked. However, above 75% capacity, adaptive routing gives a higher latency than deterministic routing. This is because dimension-order routing reduces contention by concentrating most of the traffic on the through channels of each switch [9]. With adaptive routing, the switch traffic is more uniform, resulting in higher latency.

Dynamic adaptive routing outperforms static adaptive routing at high traffic levels. The dynamic algorithm allows more flexible buffer assignments allowing packets to make progress that would otherwise be blocked waiting on a particular buffer.

Adaptive routing gives a significant performance advantage for traffic patterns that load channels non-uniformly. Figure 6 shows latency as a function of offered traffic for three

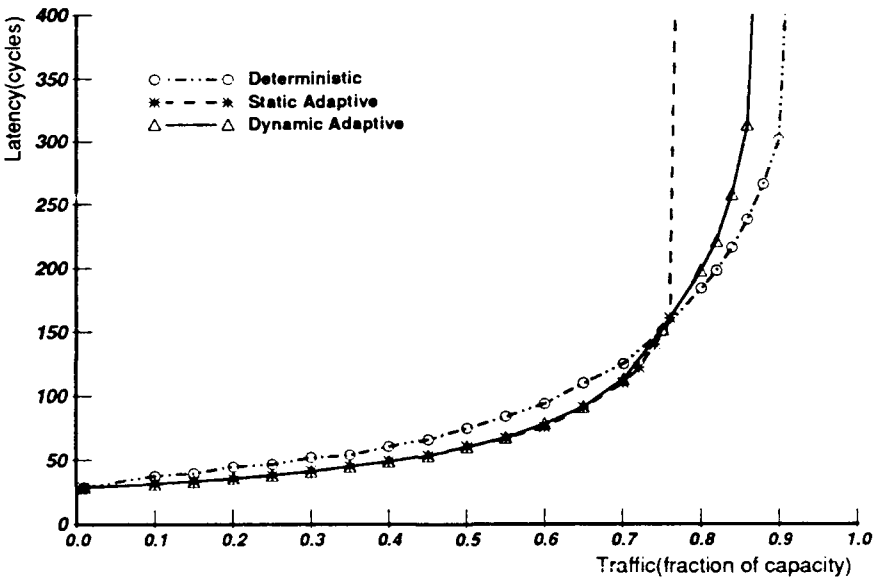

Figure 5: Latency vs. accepted traffic for a 16-ary 2-cube under random traffic. Deterministic, dimension-order routing is compared with static and dynamic adaptive routing. With random traffic, adaptive routing gives slightly lower latency with low traffic than deterministic routing but saturates first.

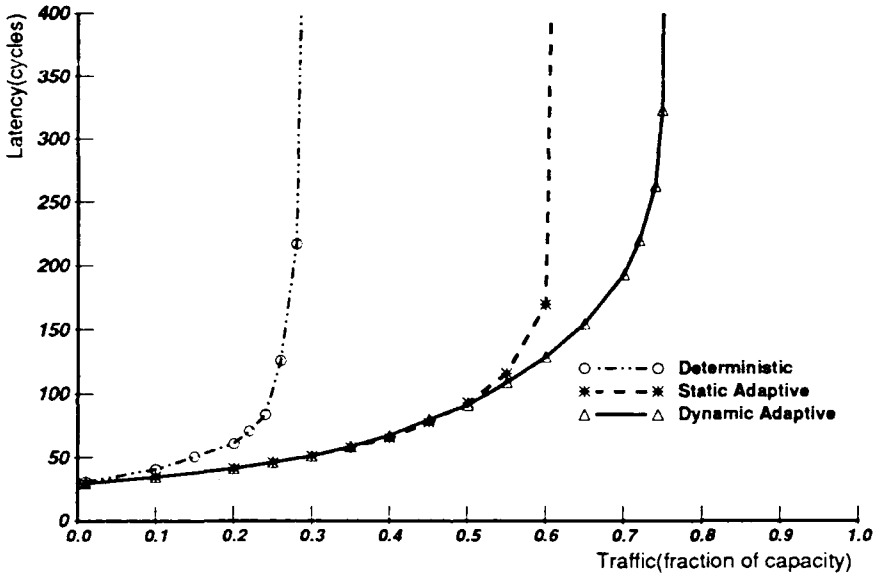

Figure 6: Latency vs. accepted traffic for a 16-ary 2-cube under bit reversal traffic. Deterministic dimension order routing is compared with static and dynamic adaptive routing. This non-uniform traffic pattern causes deterministic routing to perform very poorly, saturating at about 25% capacity. Static and dynamic adaptive routing achieve three times this performance (saturating at 60% and 75% capacity respectively) by routing to distribute the network load.

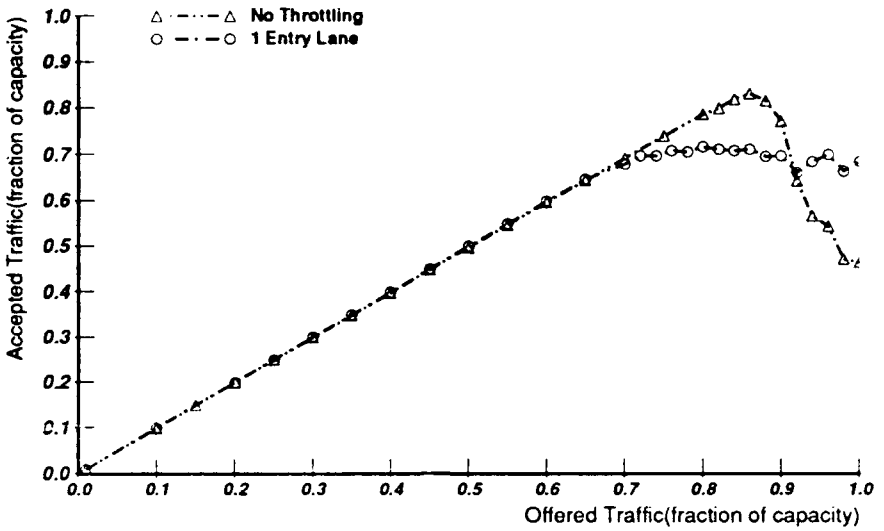

Figure 7: Throughput as a function of offered traffic for a 16-ary 2-cube network using dynamic adaptive routing with varying degrees of throttling. Throttling reduces peak throughput by restricting the entry of new packets into the network.

routing strategies under bit-reversal traffic. In this traffic pattern, each node, i, sends messages to node j where j is the bit-reversal of i. For example, node 43_{16} sends messages to node $C2_{16}$. Deterministic routing gives very poor performance for this traffic pattern, saturating at 25% capacity. This saturation occurs because a few channels become bottlenecks as in Figure 1A. With adaptive routing, packets are able to route around bottleneck channels achieving over three times the performance of deterministic routing. The static algorithm saturates at 60% capacity while the dynamic algorithm saturates at 75% capacity.

4.2 Throttling

Figure 7 shows the effect of throttling on network throughput. The figure shows throughput (accepted traffic) as a function of offered traffic for a network using dynamic adaptive routing. The simulations were run using random traffic. The curves correspond to no throttling and throttling by restricting packets with a DR of 0 (entry packets) to use only a single virtual channel of each physical channel.

Without throttling, a network using dynamic adaptive routing is unstable. As offered traffic is increased beyond 86% capacity, throughput is decreased. At saturation (all

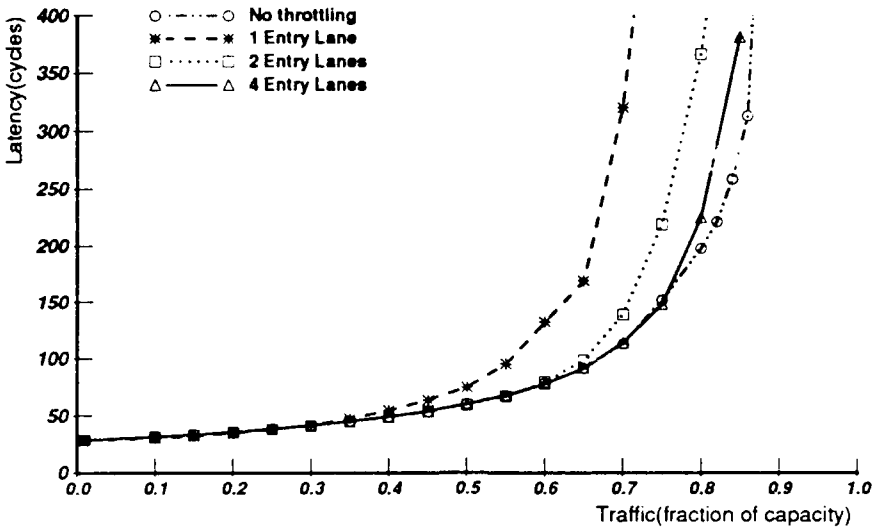

Figure 8: Throughput as a function of accepted traffic for A 16-ary 2-cube network using dynamic adaptive routing with varying degrees of throttling. Throttling increases latency by restricting the entry of new packets into the network.

sources sending all the time), throughput is reduced to 11% capacity.

Restricting the number of channels available to entry packets reduces the peak throughput from 86% capacity (no throttling) to 71% capacity with entry packets restricted to a single virtual channel. Throttling degrades throughput because the lane restrictions force some entry packets to block or turn when they would otherwise be able to make progress. However, throttling stabilizes the network. Throughput does not significantly decrease as offered traffic is increased. With throttling, saturation throughput is 66% .

Figure 8 shows the affect of throttling on latency. The figure shows the average packet latency as a function of accepted traffic for a network loaded with random traffic using dynamic adaptive routing. Curves are shown for no throttling and for throttling with 1, 2, and 4 entry channels. The figure shows that throttling increases latency as the curves for throttling approach their lower throughput asymptotes. Most of this added latency is experienced in the queue at the source node as less traffic is allowed into the network.

Figures 7 and 8 show that throttling slightly degrades latency but stabilizes the network at high traffic rates. Another advantage of throttling is that it reduces the effect of high-traffic sources on low-traffic sources. By restricting the entry of packets from high-traffic sources into the network, throttling reduces congestion giving the low-traffic sources lower

Entry Lanes	Throughput	Percent Deterministic
1	.662	0.09
2	.716	1.35
4	.339	13.0
No Throttling	.110	69.1

Table 1: Throttling reduces the number of packets forced to route on the deterministic set of virtual channels. This table shows the throughput and fraction of deterministically routed messages for various degrees of throttling under a saturation load of random traffic.

and more predictable latency.

Table 1 shows the throughput and fraction of messages that are forced into determinisitc routing for varying degrees of throttling under a random saturation source. With a saturation source, each network node attempts to inject a message into the network on each cycle. This level of traffic quickly congests all network buffers if throttling is not employed.

The table illustrates two advantages of throttling. First, the number of messages forced to route on the deterministic virtual channels is reduced. This improves fault tolerance since once a packet begins deterministic routing, it is vulnerable to a single channel fault. Second, saturation throughput is increased by throttling. Without throttling, at saturation traffic all channel buffers quickly become filled with blocked messages and the performance of the network severely degrades. Throttling restricts the traffic entering the network to a level that keeps sufficient buffers available to achieve a high throughput.

The table and figures suggests that throttling with two entry lanes offers a good compromise between latency and stability.

4.3 Selection Function

Figure 9 compares the performance of different selection functions in handling non-uniform traffic. The simulation was run with random traffic. The figure shows that minimum-congestion and straight-line selection functions give good performance for this traffic pattern. Maximum-flexibility routing results in higher latency and saturates at a lower traffic level than the other two functions. The maximum flexibility selection function causes messages to alternate dimensions once a *diagonal* to the destination is reached. This dimension alternation results in high DR numbers and a large number of packets resorting to deterministic routing. Minimum dimension reversal routing gives

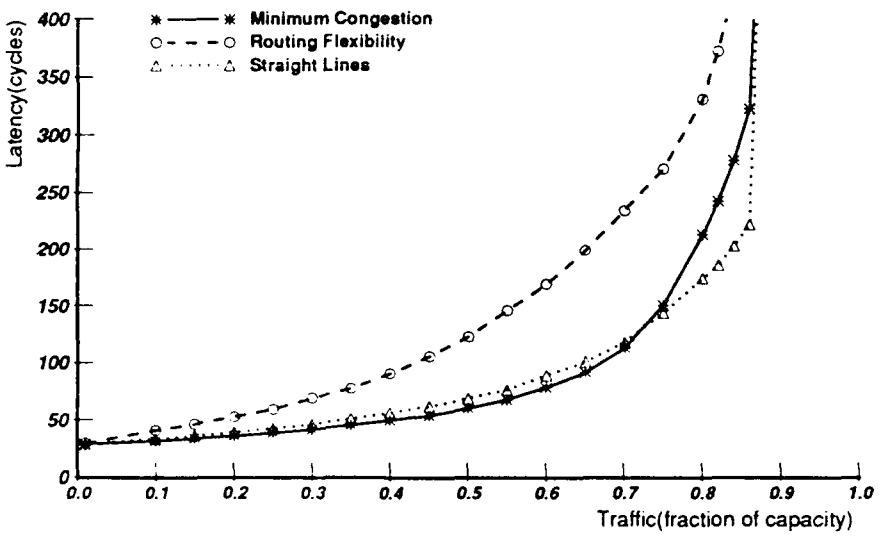

Figure 9: Latency as a function of offered traffic for four selection functions. Simulations were performed using dynamic adaptive routing and traffic from one row of a matrix transpose.

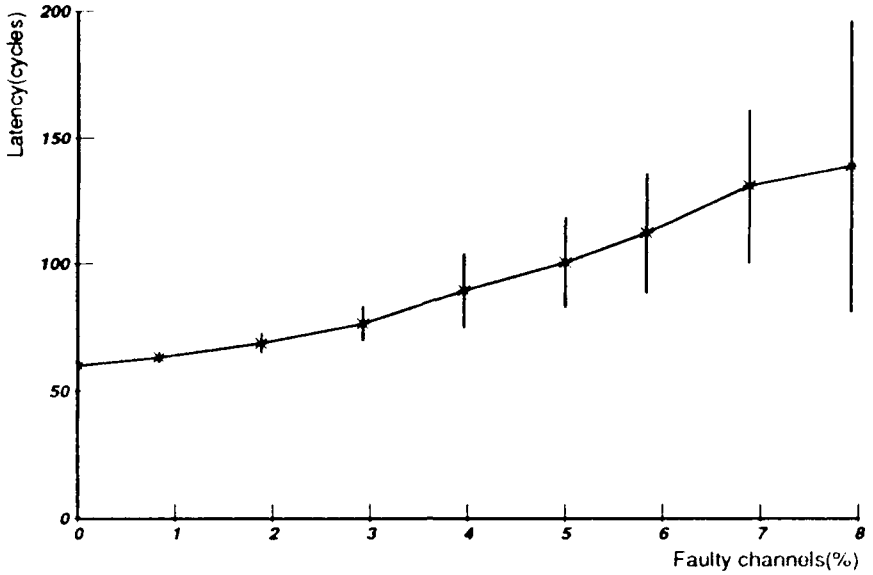

Figure 10: Latency vs. number of faulty channels for a 16-ary 2-cube network operating at 50% capacity with random traffic.

high latency for mid-range traffic levels. This selection function will not begin to adapt until all virtual channels in a given direction are full. Latency is introduced by filling these virtual channels rather than routing over idle physical channels.

4.4 Fault Tolerance

Figure 10 illustrates the graceful degradation of an adaptive network as channels fail. The figure shows the latency of the network at 50% capacity for random traffic as a function of the percentage of faulty channels. For each percentage, 20 networks were simulated, each with a different randomly chosen fault set. The asterisks in the figure gives the ensemble average latencies. The ends of the vertical error bars represent the 1σ points of each latency distribution. Latency increases only by a factor of 2.3 from a fault-free network to a network with 7.92% faulty channels (38 faulty channels in a 16-ary 2-cube). In contrast, in a network with deterministic routing, a single faulty channel renders the network inoperable.

5 Conclusion

Adaptive routing improves the performance and reliability of a multicomputer interconnection network by routing packets around congested or faulty channels. This paper has described two deadlock-free adaptive routing algorithms. Both algorithms permit misrouting and avoid deadlock by allocating virtual channels according to the number of dimension reversals a packet has made.

In the static algorithm, there is a fixed mapping between number of dimension reversals and virtual channels. This fixed assignment gives an acyclic channel dependency graph. The dynamic algorithm allows more flexible channel allocation by allocating virtual channels based on occupation to prevent cycles in the packet wait-for graph. In this case, the channel dependency graph is cyclic and adaptive routing is required to avoid deadlock.

Simulation experiments show that adaptive routing significantly improves throughput for non-uniform traffic patterns but has little effect on performance with random traffic. Adaptive routing improves throughput by a factor of three for bit-reversal traffic in a 16-ary 2-cube network. This traffic pattern causes non-uniform channel loads when dimension-order deterministic routing is employed. Adaptive routing routes around congested channels to balance the load. With random traffic, channels are loaded uniformly, and load balancing is not required.

Throttling is required to stabilize the dynamic algorithm at high traffic rates. Throttling is easily implemented by restricting new packets to route on a small number of virtual channels, *entry lanes*, until their first dimension reversal. Throttling slightly increases latency for uniform loads but reduces the affect of a hot-spot node on the network latency seen by other nodes. Throttling also reduces the fraction of messages that are forced by resource constraints to resort to deterministic routing. Simulations suggest that throttling with two entry lanes effectively stabilizes the network with only a small affect on latency.

The adaptive routing algorithms presented here can be used in conjunction with many different selection functions. Simulations show that minimum-congestion and straight-line selection functions give good performance. The maximum-flexibility selection function results in higher latency and lower throughput because it forces packets onto network diagonals.

The performance of networks using adaptive routing gracefully degrades as channels fail. Experiments show that with 7.9% of the channels faulty, latency increases by a factor of 2.3 .

With virtual channel flow control[11] and adaptive routing, multicomputer networks achieve performance approaching 90% of their physical capacity. This performance is affected little by non-uniform traffic patterns and degrades gracefully with channel fail-

ures.

The use of adaptive routing and virtual channels motivates the use of synchronous router design. Many early routers were asynchronous or self-timed to achieve maximum performance [12, 14, 15]. With deterministic routing, the design of such routers was straightforward as each dimension could operate independently and no concept of global time was required. To make use of virtual channels, however, the router must maintain timers to distinguish between a blocked channel and one that is waiting for an acknowledgement. In a synchronous router, such timing is implicit. Adaptive routing requires that information from many output channels be collected together to make a routing decision. In an asynchronous router, collecting this information poses a high synchronization overhead.

The application of these high-performance networks extends beyond connecting the processing nodes of multicomputers. Low-dimensional k-ary n-cube networks can also be used as data switches in a local-area or long-haul network and as a general-purpose backplane to connect components of digital systems. They offer a scalable alternative to buses for general-purpose interconnection in digital systems.

Acknowledgment

We thank the members of the MIT Concurrent VLSI Architecture group for their help with and contributions to this paper. Thanks also to Lisa Sardegna for proofreading this manuscript.

References

[1] J.K. Annot and R.A.H. van Twist. A Novel Deadlock Free and Starvation Free Packet Switching Communication Processor. In *Proceedings of PARLE 87*, pages 68–85, Eindhoven, The Netherlands, 1987. Philips Research Laboratories.

[2] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, 21(8):9 - 24, August 1988.

[3] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H.T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P.S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWARP: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–338. IEEE, November 1988.

[4] Andrew Andai Chien. Congestion Control in Routing Networks. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, October 1986.

[5] E. Chow, H. Madan, and J. Peterson. A Real-Time Adaptive Message Routing Network for the Hypercube Computer. In *8th Real Time Systems Symposium*, pages 88–96. IEEE, 1987.

[6] Ametek Corporation. Ametek 2010 Product Announcement, 1987.

[7] William Dally and et. al. The J-Machine: A Fine-Grain Concurrent Computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153. August 1989.

[8] William J. Dally. Fine-Grain Message Passing Concurrent Computers. In *Proceedings of the Third Conference on Hypercube Concurrent Computers*, volume 1, pages 2–12, Pasadena, CA, January 1988.

[9] William J. Dally. Network and Processor Architecture for Message-Driven Computers. In Suaya and Birtwhistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann, 1990.

[10] William J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, C-39(6):775–785, June 1990.

[11] William J. Dally. Virtual Channel Flow Control. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 60–68, May 1990.

[12] William J. Dally and Charles L. Seitz. The Torus Routing Chip. *Distributed Computing*, 1:187–196, 1986.

[13] William J. Dally and Charles L. Seitz. Deadlock Free Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–53, May 1987.

[14] William J. Dally and Paul Song. Design of a Self-Timed VLSI Multicomputer Communication Controller. In *Proceedings of the International Conferencce on Computer Design*, pages 230–234, October 1987.

[15] C.M. Flaig. VLSI Mesh Routing Systems. Master's thesis.

[16] BBN Advanced Computers Incorporated. Butterfly Parallel Processor Overview. BBN Report No. 6148, March. 1986.

[17] John N. Mailhot. Routing and Flow Control Strategies in Multiprocessor Networks. Massachusetts Institute of Technology, SB Thesis, May 1988.

[18] W.G.P. Mooij and A. Ligtenberg. Architecture of a Communication Network Processor. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Lecture Notes in Computer Science Number 365: Proceedings of PARLE '89*, pages 238–250. Springer-Verlag, 1989.

[19] John Y. Ngai and Charles L. Seitz. A Framework for Adaptive Routing in Multicomputer Networks. In *ACM Symposium on Parallel Algorithms and Architectures*. 1989.

[20] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[21] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, second edition, 1988.

To appear in Proc. ICCAD-90

Parallel Simulation Algorithms for Grid-Based Analog Signal Processors

L. Miguel Silveira Andrew Lumsdaine Jacob K. White

Research Laboratory of Electronics
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

In this paper, specialized algorithms for circuit-level simulation of grid-based analog signal processing arrays on a massively parallel processor are described and implementation results presented. In our approach, the trapezoidal rule is used to discretize the differential equations that describe the analog array behavior, Newton's method is used to solve the nonlinear equations generated at each time-step, and a block conjugate-gradient squared algorithm is used to solve the linear equations generated by Newton's method. Excellent parallel performance of the algorithm is achieved through the use of a novel, but very natural, mapping of the circuit data onto the massively parallel architecture. The mapping takes advantage of the underlying computer architecture and the structure of the analog array problem. Experimental results demonstrate that a full-size Connection Machine can provide a 1400 times speedup over a SUN-4/280 workstation.

1 Introduction

The recent success using one and two dimensional resistive grids to perform certain filtering tasks required for early vision [Mead 88] has sparked interest in general analog signal processors based on arrays of analog circuits coupled by resistive grids. As is usually the case, before fabricating these analog signal processors, substantial circuit-level simulation must be performed to insure correct functionality. Although desirable, simulation of *complete* signal processors has not been attempted because of the computational cost. Ambitious circuits consist of arrays of cells where the array size can be as large as 256×256, and each cell may contain up to a few dozen devices [Wyatt 88]. Therefore, simulation of a complete signal processor requires solving a system of differential equations with ***hundreds of thousands*** of unknowns.

The structure of grid-based analog signal processors is such that they can be simulated quickly and accurately with specialized algorithms tuned to certain parallel computer architectures. In particular, the coupling between cells in the analog array is such that a block-iterative scheme can be used to solve the equations generated by an implicit time-discretization scheme, and furthermore, the regular structure of the problem implies that the simulation computations can be accelerated by a massively parallel SIMD computer, such as the Connection Machine ®[1] [Hillis 85]. In the next section of this paper, we describe an example grid-based analog signal processor, and in Section 3 we describe the simulation algorithm. In Section 4, the mapping onto the Connection Machine is detailed. Experimental results are presented in Section 5 and the conclusion and suggestions for further work are contained in Section 6.

[1] Connection Machine is a registered trademark of Thinking Machines Corporation.

2 Problem Description

Consider the circuit in Figure 1, an idealized version of a grid-based analog signal processor used for two-dimensional image smoothing and segmentation [Lumsdaine 90]. The node equation for a grid point i,j in the network is

$$\begin{aligned} c\dot{v}_{i,j} &= g_f(v_{i,j} - u_{i,j}) \\ &\quad + g_s(v_{i,j} - v_{i+1,j}) + g_s(v_{i,j} - v_{i-1,j}) \\ &\quad + g_s(v_{i,j} - v_{i,j+1}) + g_s(v_{i,j} - v_{i,j-1}) \end{aligned} \tag{1}$$

where $u_{i,j}$ represents the image data at the grid point i,j, $v_{i,j}$ is the output voltage at node i,j, g_f is the input source impedance, c is the parasitic capacitance from the grid node to ground, and $g_s(\cdot)$ is a nonlinear "fused" resistor. In this circuit, the g_s resistors pass currents in such a way as to force $v_{i,j}$ to be a spatially smoothed version of $u_{i,j}$, unless the difference between neighboring $u_{i,j}$'s is very large. In that case, g_s no longer conducts, there is no smoothing, and the image is said to be "segmented" at that point.

In a more complete representation of the image smoothing and segmentation circuit, the voltage source $u_{i,j}$ and the source impedance g_f are replaced with a subcircuit which typically contains operational amplifiers and a phototransistor. If such a subcircuit has M internal nodes and contains only voltage-controlled elements, then it can be described by a differential equation system of the form

$$\frac{d}{dt} q_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t) = f_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t) \tag{2}$$

where $\tilde{v}_{i,j} \in \Re^M$ is the vector of the i,j^{th} subcircuit's internal node voltages, and $q_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t)$, $f_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t) \in \Re^m$ are the vectors of sums of charges and sums of resistive currents, respectively, at each of the subcircuit's internal nodes. Incorporating the subcircuit's behavior into the equation for grid point i,j leads to

$$\begin{aligned} c\dot{v}_{i,j} &= i_{sub}(v_{i,j}, \tilde{v}_{i,j}) \\ &\quad + g_s(v_{i,j} - v_{i+1,j}) + g_s(v_{i,j} - v_{i-1,j}) \\ &\quad + g_s(v_{i,j} - v_{i,j+1}) + g_s(v_{i,j} - v_{i,j-1}), \end{aligned} \tag{3}$$

where $i_{sub}(v_{i,j}, \tilde{v}_{i,j})$ is the current entering subcircuit i,j from grid node i,j.

For our purposes, an $N \times N$ grid-based analog signal processor, or analog array, is any circuit that can be described by a system of equations generated by replicating Eqn. (2)

and Eqn. (3) for each $i, j \in 1, \ldots, N$. Note that this definition enforces a regular structure, and only allows for coupling between neighboring subcircuits through two-terminal nonlinear resistors. The nonlinear resistors are usually implemented with transistors, so our definition of an analog signal processor still represents an idealization, although the extension to the general case is straightforward.

3 Numerical Algorithms

For notational simplicity, the system of equations that describe an $N \times N$ grid-based analog signal processor, defined in the previous section, will be written compactly, and perhaps not very informatively, as

$$\frac{d}{dt}q(v) = f(v(t)), \tag{4}$$

where $q(v(t)), f(v(t)) \in \Re^{N^2 \times (M+1)}$ are the vectors of sums of node charges and node resistive currents.

The transient simulation of the analog grid involves numerically solving (4). To compute the solution, it is possible to use simple explicit or semi-implicit numerical integration algorithms, but for these types of circuits, experiments show that an implicit method like the trapezoidal rule is substantially more efficient [Silveira 90]. The trapezoidal rule leads to the following algebraic problem at each time step h:

$$q(v(t+h)) - q(v(t)) + \frac{1}{2h}[f(v(t+h)) + f(v(t))] = 0. \tag{5}$$

As is standard, the algebraic problem is solved with Newton's method,

$$J_F(v^m(t+h))[v^{m+1}(t+h) - v^m(t+h)] = -F(v^m(t+h)) \tag{6}$$

where

$$F(v(t+h)) = [q(v(t+h)) - q(v(t))] + \frac{1}{2h}[f(v(t+h)) + f(v(t))] \tag{7}$$

and the Jacobian $J_F(v(t))$ is

$$J_F(v(t+h)) = \frac{\partial q(v(t+h))}{\partial t} + \frac{1}{2h}\frac{\partial f(v(t+h))}{\partial t} \tag{8}$$

In "classical" circuit simulators such as SPICE [Nagel 75], the linear system of equations for each Newton iteration is solved by some form of sparse Gaussian elimination. When simulating grid-based signal processors, where the coupling between subcircuits is restricted to nonlinear resistors, the Newton iteration equation will be such that its solution can be efficiently computed by iterative algorithms like conjugate-gradient squared (CGS) [Sonneveld 89, Burch 89]. To demonstrate this, in Table 1, we compare the CPU time required to compute the transient analysis of the network in Figure 1 using several different matrix solution algorithms to solve the Newton iteration equation. This problem is hard for an iterative method because, though not described here, the transient analysis is performing a continuation on the nonlinear resistor elements that changes the conditioning of the matrix with time (see [Lumsdaine 90] for details). As the table indicates, sparse Gaussian elimination is much slower than CGS[2] or ILU preconditioned CGS, both of which perform almost identically. This is a fortunate result, because our goal is to develop an efficient parallel simulator, and unpreconditioned CGS is easiest to parallelize.

[2] This problem is symmetric, so CGS and standard conjugate-gradient are equivalent

Size	Direct	CG	ILUCG
16×16	16.53	11.72	10.27
32×32	156.57	60.72	50.75
64×64	1856.12	272.30	224.12

Table 1: Comparisons of serial execution time for direct, CG, and ILUCG linear system solvers when used for the transient simulation of the circuit in Figure 1, where $g_f = 3.0e-5$ and g_s has a conductance of $1e-3$ when linearized about zero.

4 Connection Machine Implementation

The Connection Machine model CM-2 is a single-instruction multiple data (SIMD) parallel computer consisting of 65,536 bit-serial processors and 2048 Weitek floating-point processors. The bit-serial processors are clustered together into groups of 16 to make a single integrated circuit, and these IC's are connected together in a 12-dimensional hypercube. Two IC's, or 32 processors, share a single Weitek IC. Since the CM-2 contains 2048 Weitek IC's, a speedup of a factor of 2048 over conventional computers containing a single Weitek IC (e.g., a SUN-4) is conceivable.

For an algorithm to approach this peak parallel performance on the CM, it must satisfy three requirements. First, the problem must have enough parallelism to use all the available processors. Second, the algorithm can depend only on local or infrequent interprocessor communication, like on any parallel machine. And third, the algorithm must be mostly *data-parallel* because of the SIMD nature of the Connection Machine. By data-parallel we mean:

- One can identically map individual pieces of data to individual processors for all relevant processors and
- One can operate identically on the data with all the relevant processors

The general circuit simulation problem violates all three of the above constraints, and previous attempts at circuit simulation on the Connection Machine have not yielded impressive results [Webber 87, Silveira 90]. As we will show in the rest of this section, simulation of grid-based analog signal processors *is* well suited to the CM. These circuits are large, and can be simulated with algorithms that are mostly data-parallel and which depend on mostly nearest-neighbor communication between processors.

4.1 Data to Processor Mapping

The two-dimensional nature of grid-based analog signal processing circuits naturally maps into a two-dimensional geometry on the CM, in such a way as to maintain data parallelism and locality. The circuit is divided into identical cells (as shown in Figure 1) and each processor is assigned the data associated with each cell, with nearest-neighbor grid cells being mapped to nearest-neighbor processors. It is an important point that the assigned data includes the node voltages, currents, charges and derivatives, ***but not*** a complete description of the cell, only the CM's front-end computer has that.

It can be seen from Figure 1 that some elements in each cell cross the cell boundaries, and the communication so implied must be organized carefully to maintain maximum data-parallelism. In our approach, copies are made of ***shared-nodes***,

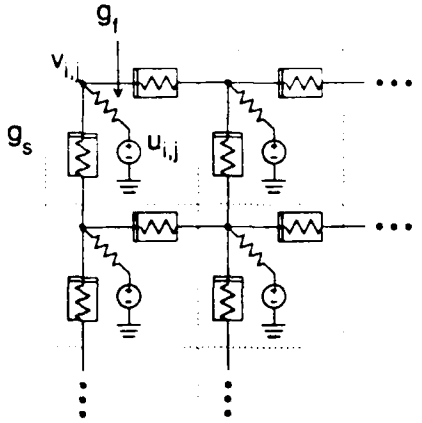

Figure 1: Grid of nonlinear resistors and its division into identical cells.

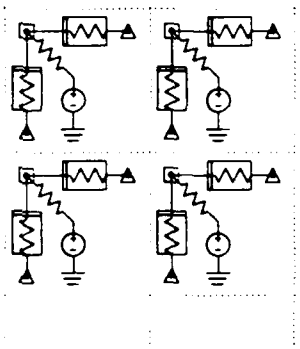

Figure 2: Separation of the cells by duplicating nodes. The shared nodes and pseudo-nodes are outlined with squares and triangles, respectively.

by which we mean nodes within each cell to which elements from other cells are connected. These copies are referred to as *pseudo-nodes*. As can be seen in Figure 2, using pseudo-nodes implies that the data for the cell devices is contained completely within the cell.

Two types of consistency between the shared-nodes and pseudo-nodes must be maintained through interprocessor communication, namely:

Voltage Consistency: Pseudo-Nodes must have the same voltage as their corresponding shared nodes.

Charge and Current Consistency: Charges and currents flowing into the pseudo-nodes are summed at the corresponding shared nodes.

This particular mapping of the circuit data insures that the many cells in a large grid can be simulated in a data-parallel fashion. That is, simulation of the entire grid is accomplished by simulating a simple cell using many copies of data, and then enforcing the voltage, charge, and current consistencies for the shared nodes and pseudo-nodes. Note that the cells on the east and south circuit boundaries respectively do not have east and south connecting elements. In order to model this properly on the CM, boundary processors are turned off whenever data corresponding to non-existent elements is manipulated. For reasons of clarity we will omit further discussion of this operation; it is performed in a straightforward manner for all the algorithms to be presented

4.2 Device Evaluation

Evaluating the right-hand side and the Jacobian for the Newton iteration, equation (7), involves computing sums of device currents and charges. Given the previous discussion of the data to processor mapping, the device evaluation portion of the simulation is obvious:

1. Copy node voltages from shared nodes to pseudo-nodes (voltage consistency step)
2. Evaluate cell devices in parallel
3. Sum node charges and currents from pseudo-nodes to shared nodes (charge and current consistency step)

4.3 Linear System Solution

As mentioned in the previous section, for the case of grid-based analog circuits, solving the linear Newton iteration equation (6) using CGS is not only easy to parallelize, it is faster than using sparse Gaussian elimination, and nearly as fast as using ILUCGS. There are two parts of the CGS iteration which involve parallel data: the vector inner product and the matrix-vector product. The vector inner-product is accomplished with an in-place multiply and a global sum. The matrix-vector product $y = Ax$ is accomplished with the following sequence of operations:

1. Copy x values from shared nodes to pseudo-nodes (voltage consistency step)
2. Perform matrix-vector product with simple-cell matrix
3. Sum y values from pseudo-nodes to shared nodes (current consistency step)

That the operations involved in the matrix vector product are similar to those required for the device evaluation should come as no surprise. The communication steps are still required for consistency, and the device evaluation step is now replaced by an in-place matrix-vector product where the local matrix corresponds to the linearized conductance matrix of the simple cell circuit.

5 Experimental Results

In order to test our algorithms, a simulation program was written for the CM, using MIT's SIMLAB program [Simlab] as a base. The parallel portions of the code were written in C* Version 6.0, a CM superset of C. The front-end experiments were run on a conventional SUN-4 workstation and the CM results were obtained on a 16K CM-2 with double-precision floating point hardware. All computations were performed in double precision arithmetic.

In Table 2, the CPU times required on the CM and the SUN-4/280 to perform the DC and transient analysis of the nonlinear resistive grid of Figure 1 are compared. Only a 16k processor CM was available, but using the "virtual processor" feature of the CM, the 256 x 256 example was simulated as if the CM had 64K processors. A real 64K machine would have run the 256x256 example approximately four times faster, and have produced simulation results approximately 1400 times faster than a SUN-4/280 workstation.

To investigate how well simulation of more realistic circuits can be accelerated, the CM simulator was tested on an idealization of the Retina chip [Mead 88]. To generate the Retina-like circuit, the voltage source $u_{i,j}$ and the source impedance

Size	DC		Trans	
	Serial	CM	Serial	CM
64×64	147.47	3.96	268.47	6.28
128×128	632.80	3.99	7581.70	44.99
256×256	(2710.2)	10.84	(214110.4)	610.27

Table 2: Comparisons of serial and CM execution times (using CGS). Extrapolated serial results are contained in parentheses.

g_f for the circuit in Figure 1 are replaced with the Retina chip's follower-connected transconductance amplifier subcircuit. This subcircuit has eight internal nodes and consists of twelve MOS devices (SPICE MOS level 3 is used in SIMLAB). The coupling resistors, represented by g_s in Figure 1, were set a small value, $10K\Omega$, and a large value, $10M\Omega$, to test the simulator. The run times on the CM and the SUN-4 for simulating these two examples are given in Table 3. For the largest example shown, the best CM algorithm is over 285 times faster than the best serial algorithm.

There are two columns of CM results in Table 3, corresponding to two types of CGS algorithms. In the first column, the times using an unpreconditioned CGS algorithm are given, and in the second column, the times using a block-preconditioned CGS (CGSB) algorithm is given. The block preconditioner used here is easy to compute in parallel, it only involves factoring each of the subcircuit Jacobians and corresponds to solving that piece of the system described by equation (2) directly. As can be seen from the table, this preconditioner accelerates the CGS convergence enough to improve the CM run times by as much as a factor of four.

Circuit		Serial		CM	
g_s	Size	Direct	CGSIL	CGS	CGSB
1e4	4×4	26.00	28.92	201.03	139.37
	8×8	106.22	121.25	452.95	143.91
	16×16	435.38	496.70	604.89	143.19
	32×32	1991.33	2156.97	555.22	144.31
	64×64	(9107.9)	(9366.8)	623.33	144.43
	128×128	(41657.4)	(40676.5)	589.96	142.69
1e7	4×4	26.73	28.17	140.64	121.74
	8×8	106.03	113.98	142.28	121.81
	16×16	435.37	461.92	141.98	122.22
	32×32	2043.18	1954.42	150.86	150.73
	64×64	(9588.6)	(8269.3)	141.69	127.89
	128×128	(44999.6)	(34988.1)	144.51	122.37

Table 3: Comparisons of serial and CM execution times. Extrapolated serial results are contained in parentheses.

6 Conclusion

In this paper, we presented algorithms for simulating very large grid-based analog VLSI circuits on a massively parallel processor. By restricting our attention to this class of circuit problems, we were able to more fully exploit the capabilities of the Connection Machine, as demonstrated by the experimental results. It is perhaps remarkable to note that with our implementation on the CM, we were able to simulate a realistic vision circuit with almost 200,000 devices and 150,000 nodes in less than 10 minutes.

Our future work will be to extend the simulator to allow more general cell interconnection and to investigate whether further speed improvements can be obtained through the use of nonlinear Krylov subspace methods — the nonlinear analogue of CG.

Acknowledgments

This work was supported by the Defense Advanced Research Projects Agency under Contract No. N00014-87-K-825 and by the Portuguese INVOTAN committee. The authors are grateful to Thinking Machines Corporation, especially Rolf Fiebrich, for providing hardware and software support for the development of the simulator. The authors would also like to thank Prof. John Wyatt and the MIT Vision Chip Project group for providing the motivation for this work.

References

[Burch 89] R. Burch, K. Mayaram, J.-H. Chern, P. Yang, P. Cox, "PGS and PLUCGS - Two New Matrix Solution Techniques for General Circuit Simulation," *Proc. ICCAD-89*, pp. 408 - 411, Nov. 1989.

[Hillis 85] W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.

[Lumsdaine 90] A. Lumsdaine, J. Wyatt, and I. Elfadel, "Nonlinear Analog Networks for Image Smoothing and Segmentation," *Proceedings of the International Symposium on Circuits and Systems*, pp. 987 - 991, May, 1990.

[Mead 88] C. A. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley, Reading, MA, 1988.

[Nagel 75] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Electronics Research Lab Report, ERL M520, Univ. of Calif., Berkeley, May 1975.

[Silveira 90] L. M. Silveira, "Circuit Simulation Algorithms for Massively Parallel Processors," S. M. Thesis, MIT, May 1990.

[Simlab] A. Lumsdaine, M. Silveira, and J. White, "Simlab Programmer's Guide," To be published as an MIT memo.

[Sonneveld 89] P. Sonneveld, "CGS, A Fast Lanczos-type Solver for Nonsymmetric Linear Systems," SIAM J. Sci. Stat. Comp., 10(1989), pp. 36-52.

[Tong 88] C. Tong, "The Preconditioned Conjugate Gradient on the Connection Machine," *Proceedings of the Conference on Scientific Applications on the CM*, Horst D. Simon, ed., pp. 188-213, World Scientific, Singapore, 1988.

[Webber 87] D. M. Webber, A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine," *24th ACM/IEEE Design Automation Conf.*, pp. 108-113, June 1987.

[Wyatt 88] J. Wyatt, et al, *Smart Vision Sensors: Analog VLSI Systems for Integrated Image Acquisition and Early Vision Processing*, Proposal, MIT, 1988.